Computer Science

Constrained Objects

Bruce Horn

28 November 1993
CMU-CS-93-154

Carnegie Mellon

Constrained Objects

Bruce Horn

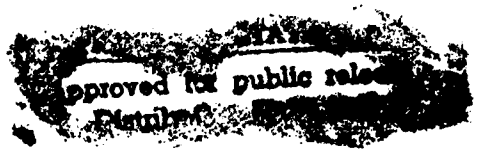
28 November 1993
CMU-CS-93-154

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

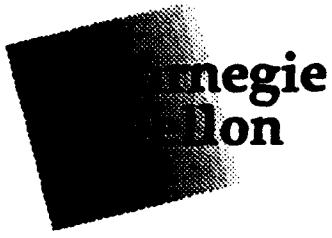
Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Thesis Committee:
James H. Morris, Co-Chair
Jeannette M. Wing, Co-Chair
Richard Rashid
Alan Borning, University of Washington
Kristen Nygaard, University of Oslo, Norway

This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597.



Keywords: programming languages, declarative programming, object-oriented programming, encapsulation, constraint solving, term rewriting.



School of Computer Science

DOCTORAL THESIS
in the field of
Computer Science

Constrained Objects

BRUCE HORN

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ACCEPTED:

Jeannette M. Wing — THESIS COMMITTEE CHAIR — 13 December 1993 — DATE

D. Clarke — THESIS COMMITTEE CHAIR — 12/13/93 — DATE

H. Clarke — DEPARTMENT HEAD — 12/12/93 — DATE

APPROVED:

R. Rashid — DEAN — 12/13/93 — DATE

Abstract

Object-oriented programming has shown to be a suitable paradigm for implementing a wide variety of systems. The concepts of encapsulation and inheritance are powerful and useful features, providing flexibility in an understandable and extensible framework. Most object-oriented programming languages, however, are complicated; in providing these features, they present a bewildering array of mechanisms for describing programs, such as classes, metaclasses, methods, class variables, and various control structures. Even more importantly, object-oriented languages do not provide explicit support for maintaining the consistency of internal object states; it is up to the programmer to code methods that respect an implicit set of consistency requirements for the object as a whole.

This thesis presents a new model of programming, called *constrained objects*, in which algebraic constraints are used as a foundation for object-oriented programming. In this model, objects are encapsulated constraint systems that communicate with each other via a message-passing mechanism. Each object maintains a consistent state under perturbation from messages sent by other objects using a constraint satisfaction process. One object may inherit from another; the set of constraints from the first object is conjoined with the set of constraints from the second, and the subsequent satisfaction of the combined constraints ensures the semantic consistency of the derived object.

We implement the constrained objects model by using a general abstraction mechanism called a *constraint pattern*, with which we build Siri, an object-oriented language. In Siri, constraint patterns play the roles of code and data abstractions, and subsume classes, instance variables, methods and control structures. Siri's constraint patterns, a conceptual blend and extension of BETA's hierarchical, block-structured patterns, and of Bertrand's augmented term rewriting rules, use equation solving for constraint satisfaction, method generation, compilation, and checking of inheritance compatibility. Siri uses explicit constraints on the state of the object to maintain internal object consistency, and provides constraint-based method definitions for more expressive coding.

A prototype version of Siri has been implemented using a small interpreter written in C; the rest of the language is defined using primitive code written in Siri itself.

Table of Contents

Chapter 1 Introduction 1

1.1 Object-Oriented Programming 1

1.1.1 Encapsulation 1

1.1.2 Inheritance 4

1.2 Constraint Programming 7

1.2.1 Types of Constraints 7

1.2.2 Constraint Satisfaction Mechanisms 8

1.3 Object-Oriented Programming with Constraints 10

1.3.1 Messages vs. Constraints 11

1.3.2 The Problem 12

1.3.3 Solution Goals 12

1.4 Thesis Contributions 13

1.5 Using the Constrained Objects Model with Siri 14

1.5.1 The Rectangle Example 14

1.5.2 Evaluating `MoveTo` Using ATR+ 15

1.5.3 Inheritance 16

1.6 Related Work 16

1.6.1 Bertrand 17

1.6.2 ThingLab and ThingLab II 18

1.6.3 Kaleidoscope 18

1.6.4 BETA 19

1.6.5 Other Related Languages 20

1.7 Roadmap to Thesis 20

Chapter 2 The Model 23

2.1 Object Interface: the Outside View 24

2.1.1 Attributes 25

2.1.2 Methods 26

2.2 Object Implementation: the Inside View 26

2.2.1 Attributes 26

2.2.2 Constraints 28

2.2.3 Methods 30

2.3 Communication Mechanisms 32
2.3.1 System Messages 32
2.3.2 A System Message Example 33
2.3.3 User Messages 35
2.3.4 Constraint Evaluation 35

2.4 Object Instantiation 36

2.5 Inheritance 37
2.5.1 Attribute Inheritance 38
2.5.2 Constraint Inheritance 39
2.5.3 Method Inheritance 40

2.6 A Simple Example 43

2.7 Implications of the Model 45
2.7.1 Separation of Side-Effecting and Non-Side-Effecting Interfaces . . 45
2.7.2 Constraint-Based Method Definition 46
2.7.3 Restricted Pointer Manipulation 47
2.7.4 Strict Top-Down Inheritance 48
2.7.5 Restricted Constraint Solving to a Single Object 49

2.8 Model Summary 50

Chapter 3 Constraint Patterns in Siri 53
3.1 Term Rewriting Mechanisms and ATR+ 55
3.1.1 Standard Term Rewriting (TR) 55
3.1.2 Augmented Term Rewriting (ATR) 57
3.1.3 ATR Extended for Constraint Patterns: ATR+ 59

3.2 The Basic Abstraction 61

3.3 Objects as Types 62
3.3.1 Type Conformance 62
3.3.2 Type Specificity 64
3.3.3 Role Types 64

3.4 Labels 65
3.4.1 Simple Labels 65
3.4.2 Parameterized Labels 65
3.4.3 Label Repetitions 67

3.5 Prefixes 69

3.6 The Pattern Body 69
3.6.1 Basic Expressions 70
3.6.2 Operators for Structuring Constraint Patterns 70
3.6.3 User-Defined Operators 71
3.6.4 `self` 72
3.6.5 Constraint Expressions 73
3.6.6 Imperative Expressions 74
3.6.7 Expression Types 76
3.6.8 Nested Patterns 76
3.6.9 Scoping Rules 77
3.6.10 Accessing an Object's Evaluation Environment 79

3.7 Uses of the Constraint Pattern 81
3.7.1 Patterns as Classes and Modules 81
3.7.2 Patterns as Functions 83
3.7.3 Patterns as Instance Variables 84
3.7.4 Patterns as Object Attributes 84
3.7.5 Patterns as Methods that Modify State 85
3.7.6 Patterns as Abstract Classes, Attributes, and Methods 87
3.7.7 Patterns as Control Structures 88

3.8 Limitations of Siri and the Constraint Pattern Abstraction 91
3.8.1 Separate Constraints and Messages 92
3.8.2 Limited Constraint Domain 92
3.8.3 No Constraints on Shared Attributes 92
3.8.4 No First-Class Constraint Patterns 93

3.9 Advantages of Siri and the Constraint Pattern Abstraction 93
3.9.1 A Single Abstraction Mechanism 93
3.9.2 Fully Encapsulated Objects 94
3.9.3 Expressive Method Coding 94

Chapter 4 A Siri Implementation 95
4.1 Evaluating a Constraint Pattern 95
4.1.1 Parsing a Constraint Pattern 96
4.1.2 Matching Reducible Expressions 98
4.1.3 The Residual 100
4.1.4 Instantiating Objects 102

4.2 Local Constraint Satisfaction 108
4.2.1 Equation Simplification Patterns 108
4.2.2 Ordered Linear Combinations 109
4.2.3 Binding and Solving for Attributes 110

4.2.4 Redundant and Conflicting Expressions . . . 113

4.3 Modifying Objects by Method Evaluation . . . 115
4.3.1 Method Reduction . . . 116
4.3.2 Method Invocation . . . 116
4.3.3 Value Dependencies . . . 117
4.3.4 Structure Dependencies . . . 118

4.4 The Object Structure . . . 118
4.4.1 Object Identifiers . . . 118
4.4.2 Basic and Extended Object Headers . . . 119
4.4.3 Objects with Basic Headers . . . 120
4.4.4 Objects with Extended Headers . . . 122

4.5 Implementation Summary . . . 125

Chapter 5 Optimizations and Improvements for Siri . . . 127
5.1 Current Performance and Capability Limitations . . . 127
5.1.1 Memory-Intensive Structures . . . 128
5.1.2 Slow Evaluation . . . 128
5.1.3 Redundant Code Reevaluation . . . 128
5.1.4 Limited Constraint Solver . . . 129

5.2 Space Optimizations . . . 129
5.2.1 Minimizing Required Header Information . . . 129
5.2.2 Sharing Expressions . . . 131
5.2.3 Computing Attribute Values . . . 132

5.3 Speed Optimizations . . . 133
5.3.1 Improving Augmented Term Rewriting in ATR+ . . . 133
5.3.2 Pattern Customization . . . 135
5.3.3 User-level Hints to the System . . . 135
5.3.4 Compilation . . . 137

5.4 Incremental Recomputation Techniques . . . 138
5.4.1 Object Equality . . . 139
5.4.2 Lazy Structure Sharing . . . 141
5.4.3 Object Decompositions . . . 141
5.4.4 Pattern Caching . . . 141
5.4.5 Evolutionary Reduction . . . 142

5.5 Solver Optimizations . . . 143
5.5.1 Extending the Solver . . . 143
5.5.2 Domain-Specific Solvers . . . 144

5.5.3 Using a Constraint Network 144

5.6 Summary of Optimizations **146**

Chapter 6 Summary of Contributions and Future Work . . . 149

6.1 Contributions **149**

6.1.1 The Programming Model: Constrained Objects 150

6.1.2 The Execution Model: ATR+ 151

6.1.3 Siri: the Proof-of-Concept Language 151

6.2 Future Work **152**

6.2.1 Changes in the Programming Model 152

6.2.3 Concurrent Evaluation 152

6.2.4 Constraint Patterns as Agents 152

6.2.5 Exploratory Programming 153

6.3 Conclusions **154**

List of Figures

Chapter 1 Introduction

1-1 A graphical rectangle . . . 3
1-2 aRectangle and MoveTo in C++ . . . 3
1-3 aGoldenRectangle, setWidth and setHeight in C++ . . . 5
1-4 Messages vs. Constraints . . . 11
1-5 aRectangle in Siri . . . 14
1-6 aGoldenRectangle in Siri . . . 16

Chapter 2 The Model

2-1 Messages and constraints in encapsulated objects . . . 24
2-2 aRectangle's interface . . . 25
2-3 aCFTemp's interface . . . 25
2-4 The parts of an object . . . 26
2-5 The desktop shared attribute . . . 28
2-6 An extended version of aRectangle . . . 29
2-7 A definition of aCFTemp . . . 30
2-8 theOutdoorTemp . . . 34
2-9 R derived from X, Y, and Z using inner . . . 42
2-10 aFurnace . . . 44
2-11 aThermostat with changed messages . . . 44
2-12 aThermostat with internal constraints . . . 45

Chapter 3 Constraint Patterns in Siri

3-1 An extended implementation of aRectangle . . . 86

Chapter 4 Siri: A Constraint Pattern Implementation

4-1 A syntax tree for the expression "from 10 to 20 by 2" . . . 97
4-2 A flattened syntax tree for the expression "from 10 to 20 by 2" . . . 97
4-3 The Match algorithm . . . 99
4-4 The PInstantiate algorithm . . . 103
4-5 The InstantiateResidual and Reduce algorithms . . . 103
4-6 The ResidualMerge algorithm . . . 105
4-7 The PrefixMerge algorithm . . . 106
4-8 Immediate objects . . . 118
4-9 Boxed objects . . . 119
4-10 Basic and extended object headers . . . 120
4-11 A list object stored in test1 . . . 121
4-12 An object with a vlfieldList holding three fields . . . 123

4-13 The string "Siri" stored in test2 124

Chapter 5 Optimizations and Improvements for Siri

5-1 The constraint graph for F - 32 = 9/5 * C 145
5-2 The three data paths for X * Y = Z 145

Chapter 6 Summary of Contributions and Future Work

No figures.

Acknowledgements

It is more than difficult to recognize everybody that made a difference to me during my years in graduate school. However, it's always worth a try.

I chose Carnegie Mellon over other schools because CMU was different: the students were friendly and enthusiastic, and everyone was happy that they were here! I have never been disappointed: the CMU School of Computer Science has always been a friendly, interesting, and challenging place to work, and I thank those who made it so: my advisors Jim Morris and Jeannette Wing, as well as my proto-advisors Bob Sproull and Alfred Spector; the amazing and wonderful Sharon Burks; and my friends and colleagues, with whom I've enjoyed many an hour of discussion. However, even in a wonderful environment like Carnegie Mellon, it is sometimes useful to spend some time talking with different people and seeing new places, and to allow the mind to roam. Kristen Nygaard and the Institutt for Informatikk in Oslo, Norway, provided the new place for me; in particular I would like to thank Kristen for his help in securing me the freedom to explore and develop Siri. Also I would like to thank all of my Norwegian friends, both in Norway and here in Pittsburgh, for their friendship over the years; I very much enjoy hearing a familiar Norwegian voice now and then. *Tusen takk alle sammen!*

A variety of people have helped me in my development of Siri. My friends John Anderson, Randy Brost, and Blake Ward provided early interest and support on the language design and implementation. John also provided me with his excellent word processor WriteNow, with which I wrote this thesis. While not the most powerful word processor, WriteNow always performed exceedingly well, and was a true pleasure to use. In addition I would like to thank Rob Chandhok, Mike Gleicher, Richard Helm, Nevin Heintze, Spiro Michaylov, and the referees for comments on earlier drafts of the OOPSLA 1992 paper, which became the foundation for this thesis.

My life would never have been as wonderful without the people of the Learning Research Group and the Xerox Palo Alto Research Center: Ted Kaehler, Alan Kay, Dan Ingalls, Adele Goldberg, Dave Robson, Steve Weyer, and many others. The people of Xerox PARC set me on course, at a very young age, to an interesting and satisfying career in computer science.

My parents have provided me with an amazing combination of freedom and opportunities to exercise that freedom; their interest and pride in me has kept me going through some not-so-easy times. I hope that they realize how much I appreciate their support. Thanks, M&D!

Finally, I would like to thank friend Christine Cuyler from whom I have learned many things, not the least of which is the appreciation of good bread!

Thesis Committee

James H. Morris (co-chair)
Jeannette M. Wing (co-chair)
Richard Rashid

Alan Borning (University of Washington)
Kristen Nygaard (University of Oslo, Norway)

Chapter 1
Introduction

This thesis presents a new model of programming, called *constrained objects*, in which algebraic constraints are used as a foundation for object-oriented programming. In this model, objects are encapsulated constraint systems that communicate with each other via a message passing mechanism. Each encapsulated object maintains a consistent state under perturbation from messages sent by other objects by a constraint satisfaction process. One object may inherit from another; the set of constraints from the first object is conjoined with the set of constraints from the second, and the subsequent satisfaction of the combined constraints ensures the semantic consistency of the derived object.

This thesis also describes an experimental programming language, Siri, that implements this model of programming. Siri uses a single abstraction mechanism, called a *constraint pattern*, to define class-like objects, instance variables, methods, and control structures.

In this chapter, we first describe some of the basic features of object-oriented programming that we are addressing with our new model: encapsulation and inheritance. We discuss the limitations that arise from these features using a traditional object-oriented viewpoint. Next, we give a general overview of constraint programming and the mechanisms typically used for constraint satisfaction. Then, we describe the problem being addressed by this thesis: how the benefits of constraints can be introduced into object-oriented programming while maintaining object encapsulation and inheritance. Finally, we summarize the contributions of this thesis, the related work, and present a roadmap of the thesis as a whole.

1.1 Object-Oriented Programming

Object-oriented programming has shown to be a very useful programming paradigm for a variety of systems. Objects have identity and persist through time. They consist of some mutable state, and a set of messages that can be sent to the object to invoke operations that change its state. Two of the most important attributes of object-oriented programming are *encapsulation*, which hides an object's implementation from clients, and *inheritance*, which is a facility for providing incremental definition of new objects from existing ones.

1.1.1 Encapsulation

An object consists of some internal state and an external interface available to clients. Generally, an object represents its state by named slots, or instance variables. Not all changes of object state may be allowed; encapsulation is a mechanism by which the object representation is kept hidden. A *message* interface provides a way for clients to observe the object's state as

well as to change it, without knowledge of the implementation.

Methods are the object's internal implementation for these messages. While the same message may be sent to different objects, different methods may be invoked to satisfy them, based on the kind of object and how it needs to handle the message. For example, a `Circle` and a `Square` might both accept a `draw` message, but would have entirely different methods for performing the operation. In this way, message-passing decouples the intent of messages from their implementation, and thus methods are responsible for maintaining the internal consistency of an object.

Furthermore, since clients may not access the internal object representation, and instead must go through the message interface, encapsulation can be achieved by protecting the object from inconsistent state changes by clients. Object clients need only concern themselves with the messages available for that object, and the parameters required. The implementation of the object can be changed at any time without affecting clients, as long as the interface continues to be supported. Thus, encapsulation separates the object's implementation from its interface, maintaining object-oriented control of the state and implementation.

An object's state may contain values or structures, stored in its instance variables. We consider these entities, if not shared outside of the object, to be encapsulated within the main object. Such an arrangement of subobjects form a *part-whole hierarchy* defining the object's state. In this case, encapsulation serves as an effect boundary, limiting the scope of state changes to within the object itself. This restriction of state changes using encapsulation is possible as long as no external objects are referenced or modified during execution of a method. In an object defined using a part-whole hierarchy, the state of the object is the union of the states of the subobjects; there is no shared state with any external object.

In an object-oriented program, when a programmer defines an object, its state, and the messages to which it responds, there is usually a mental model of how the different parts of the object relate to each other. The programmer must write each method such that a set of implicit constraints is not violated when the state is changed. The correctness of the object's implementation relies on these implicit constraints being continually satisfied as messages are received and methods invoked. Therefore, in a traditional object-oriented language, there is implicit information in the coding of an object.

To illustrate this, we describe a movable rectangle that is displayed on a screen. This rectangle has characteristics of interest to a programmer, including its `top`, `left`, `bottom`, and `right` coordinates; and its `width`, `height`, and `center` point. We would like to be able to move this rectangle by its `center` point as well.

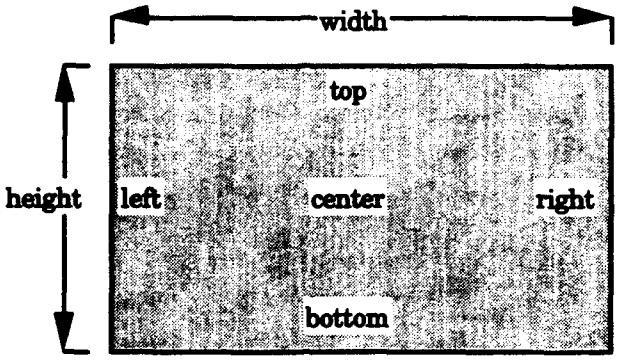


Figure 1-1. A graphical rectangle.

A programmer could implement this rectangle object in C++ as follows:

```
class aRectangle {
   public:
      float          top, left, bottom, right;  // rectangle coordinates
      float          width(void);               // return the width
      float          height(void);              // return the height
      pt             center(void);              // return the center point
      void           moveTo(pt newCenter);      // move to a new center point
};

void  aRectangle::moveTo(pt newCenter) {
   pt delta = newCenter - center();        // compute the difference between
                                           // the old and new centers, and then
   left += delta.x; right += delta.x;      // update the coordinates by adding
   top += delta.y;  bottom += delta.y;     // the appropriate values.
};
```

Figure 1-2. `aRectangle` **and** `moveTo` **in C++.**

In this case, the object is defined in a part-whole hierarchy, where the rectangle itself is defined by four instance variables that are wholly-owned parts. Since no state is shared, the rectangle is fully responsible for the consistency of its own state.

However, if we want to define internal consistency requirements that maintain relationships between object attributes, an object must be fully encapsulated, and it must also be defined in a part-whole hierarchy to avoid aliasing. In the implementation shown, the object is not fully

encapsulated: the instance variables `top`, `left`, `bottom`, and `right` are publicly modifiable. Without full encapsulation, an object's consistency requirements could be subverted via assignment to instance variables.

While there are advantages to unencapsulated objects (a programmer may be able to take advantage of an object's implementation for efficiency purposes, say), being able to directly access an object's internals violates the abstraction barrier. If this facility is available, the parts of the program where direct access is used should be as limited as possible.

Note that the code as written embodies a large number of design decisions that affect the implementation as a whole. In the rectangle's representation, the programmer explicitly decides to code its `top`, `left`, `bottom`, and `right` coordinates as state; to code its `width`, `height`, and `center` as functions of the state; and to provide an operation that moves the rectangle, modifying its state in a particular way. This information, while explicit in the code, reflects an implicit belief by the programmer that there will be many direct accesses to the coordinates, which are stored directly, and fewer accesses to the attributes `width`, `height`, and `center`, which must be computed. In addition, the code in the `moveTo` routine, which changes the object's state, captures the implicit assumptions that the `width` and `height` should not change while the `center` is moved. When `moveTo` is invoked, the object's state is kept consistent with respect to these assumptions. However, this is not immediately apparent from an inspection of the code.

This example illustrates the lack of support for directly stating the implicit specifications for internal consistency that many objects require. From a programmer's viewpoint, it would be better to simply describe the characteristics, or *attributes* of the object and how they relate to each other, rather than define explicit state and attributes as functions of that state.

1.1.2 Inheritance

Inheritance supports refinement and software reuse in object-oriented programs. Inheritance is an incremental modification mechanism that allows one object to inherit from another; what is actually inherited varies from language to language. In general, an object `X` that inherits from an object `Z` is usually considered to be "at least" a `Z` in behavior. The ability to inherit from an existing object allows the programmer to define new objects in the system not only by using them in instance variables to store state, but also by modifying and mixing the objects' specifications.

Providing inheritance has many benefits; some of the most important are listed as follows (list from [Thomsen86]):

- *Better Conceptual Modeling.* Since specialization hierarchies are very common in everyday life, direct modeling of such hierarchies makes the conceptual structure of programs (or systems) easier to comprehend.

- *Factorization.* Inheritance supports factorization of common properties of classes of objects. These properties are described only once and reused when needed. Factorization results in greater modularity and makes complicated programs easier to comprehend and maintain since redundant description is avoided.

- *Stepwise Refinement in Design and Verification.* Inheritance hierarchies support a technique where the most general classes containing common properties of different classes are designed and verified first, and then specialized classes are developed top-down by adding more details to existing classes.

- *Polymorphism.* The hierarchical organization of classes provides a basis for introduction of polymorphism in the sense that a procedure with formal parameter of class C will accept any C object as actual parameter, including instances of subclasses of C.

Using inheritance, we can extend the class `aRectangle` to specify a new rectangle where the ratio of the `width` to the `height` is always the golden ratio (`1.618...`). Then we can write methods for setting the `width` and `height` that maintain this ratio. We say that the resultant object, `aGoldenRectangle`, is *derived* from `aRectangle`.

```
class aGoldenRectangle: public aRectangle {
   public:
      void setWidth(float newWd);
      void setHeight(float newHt);
};

void aGoldenRectangle::setWidth(float newWd) {
   pt    curCenter = center();
   float newHt = newWd/1.618;
   top = curCenter.y - newHt/2; bottom = top + newHt;
   left = curCenter.x - newWd/2; right = left + newWd;
};

void aGoldenRectangle::setHeight(float newHt) {
   pt    curCenter = center();
   float newWd = newHt*1.618;
   top = curCenter.y - newHt/2; bottom = top + newHt;
   left = curCenter.x - newWd/2; right = left + newWd;
};
```

Figure 1-3. `aGoldenRectangle`, `setWidth` and `setHeight` in C++.

Two problems are evident here. The first problem is that, as before, the intent of the two methods is not explicit: they must change the `width` or `height` without changing the `center` point. While this is implicit in the code, it is not clear what the intent is without a full understanding of the inherited object `aRectangle`. The second is that the methods `setWidth` and `setHeight` count on explicit knowledge of the implementation of `aRectangle`; in particular, that `top`, `left`, `bottom`, `right` are instance variables, and that `center` is a function of these. If the implementation of `aRectangle` changes, so must the implementation of the methods in `aGoldenRectangle`, thus breaking the abstraction. This shows that while traditional inheritance is useful for extending definitions, one must still understand the implementation of the objects being inherited, a significant intellectual overhead [Snyder86].

Another problem arises when an inherited method has the same name as a method being defined in the derived object. The traditional model of inheritance (for example, in Smalltalk [Goldberg83] and C++ [Stroustrup86]) allows overriding of inherited methods with explicit calls via `super` to invoke them. There is no requirement in such a model that a method defined in a derived object have any relationship to its namesake in its ancestor. Consider the case where an object `X` inherits from another object `Z`. In the presence of method overriding, where `X` can provide a completely new definition of a method that `Z` had provided, the correctness of the combination rests on whether the method written for `X` is compatible with a set of implicit assumptions about the `X+Z` system. Thus, the burden of assuring consistency of methods, and of combined behavior with inheritance, rests with the programmer.

In another model of object-oriented programming (for example, in Simula [Dahl70] and BETA [Kristensen89]), methods defined in the derived object are combined automatically with their corresponding methods in its ancestor using the `inner` construct. Thus, when invoking a method `M` in an object `X`, the method that is run is actually a composite, created from all of the methods `M` defined in `X` and `X`'s ancestors. This is an improvement, but the combined methods must still be checked by the programmer for the desired behavior.

Thus, inheritance in object-oriented programming is useful, but it is not strong enough: there is no mechanism for ensuring consistency of objects created through inheritance. In the rectangle example, there is no explicit declaration of the constraint that its `width` is `1.618` times the `height`, which is maintained by all of the derived object's methods. Instead, the implicit constraint is distributed among the methods in a very subtle way that requires serious inspection to discover. Programmers should be able to state these assumptions explicitly in a single location, and to be able to factor out the subtle coding that currently is distributed among the methods.

The model of object-oriented programming in this thesis provides the ability to specify explicitly the consistency requirements of objects and methods, by themselves and in the presence of inheritance, and to have them be used by the system to ensure correct behavior.

1.2 Constraint Programming

Object-oriented programming is a framework for imperative, state-based programming. In contrast, constraint systems and languages [Sutherland63, Borning79, Steele80, Gosling83, Jaffar87a, Leler88, Saraswat89, Freeman-Benson90] provide a framework for declarative programming. Given a declarative specification of relationships between objects, constraint systems automatically satisfy certain classes of constraints by solving for values of objects, equations, or relations. Constraints, like messages, also decouple the intent of the programmer from the actual implementation, but at an even higher level: the system takes a set of declarative statements and produces a plan for satisfying the constraints given a particular goal.

In the programming language domain, constraints are declared relationships between objects that are maintained by the underlying system. The following is a short overview of the field of constraint programming; for an excellent survey, see [Steinbach91].

1.2.1 Types of Constraints

Constraints come in several forms, depending on the domain of discourse:

If-then rules

```
if X is a person then X has a mother and X has a father.
```

Type or domain constraints

```
Y is an integer.
```

Numeric and algebraic constraints

```
x + y = z; x^2 + y^2 < 4; z = cos(x);
```

Quantification and Scoping

```
all x, y such that x < 10 and y < 10 and x + y < 12;
```

Because constraints are free of side effects, they are independent of order. Thus, programming with constraints is quite different from programming in a traditional, imperative programming language where the order of statements has meaning. Constraints are often a natural way to specify a problem, because they focus on *what* is desired, rather than *how* the desired result should be accomplished. By not having to specify the process for computing the result, the programmer may specify code at a much more abstract level.

In addition, constraints compactly specify knowledge that can be used in many different ways. For example, the simple arithmetic statement

```
x = y + z;
```

can be used to compute x given y and z, y given x and z, or z given x and y, depending on the context of use.

A constraint can also provide an implicit specification of a large or infinite set of objects using a small, finite expression. The infinite set of all positive even numbers is trivially specified:

```
x: aNumber; x mod 2 = 0; x > 0;
```

The expressive power of constraints comes with some disadvantages, however. We can write constraints that are arbitrarily difficult to solve. The traditional constraint-language example is Fermat's Last Theorem [Borning79]:

```
x, y, z, n: anInteger; x > 0; y > 0; z > 0; n > 2; x^n + y^n = z^n;
```

the satisfiability of which is unknown (although at this writing, Professor Andrew Wiles of Cambridge University has announced a 200-page proof [Economist93]).

It is clear that constraint expressions are very powerful; hence, many statements that are easily specified using constraints can only be solved by special-purpose techniques, while some cannot be solved at all. Because of this, a constraint language must restrict the kinds of constraints that it supports [Freeman-Benson90].

A constraint language, then, is a language for specifying a set of constraints from a particular problem domain, and an underlying set of algorithms for solving those constraints to yield a set of values that satisfy them.

1.2.2 Constraint Satisfaction Mechanisms

Because of the necessity of restricting the domain of constraints that are to be solved by a constraint language, such languages usually are designed, and optimized, for a particular problem domain. Different techniques for satisfying constraint problems have been developed for a variety of domains.

Logic and Constraint Logic Programming (CLP)

Some of the earliest constraint-based languages were developed to solve finite domain problems (i.e., problems with a finite set of choices for values) such as the n-queens problem. A language like PROLOG [Clocksin81] can be considered a constraint language, since it implements the rules of logic over the Herbrand Universe and can solve certain symbolic equations. PROLOG picks an evaluation strategy that is not complete, but is practically useful: PROLOG's execution model supports backtracking, and thus PROLOG systems can search a space of possible solutions.

CLP(R) [Jaffar87a, Jaffar87b, Heintze87] integrates PROLOG-style logic programming with equational constraints over the real numbers. For efficiency reasons, CLP(R) uses a variety of solvers for different subproblems and subdomains; for example, CLP(R)'s execution engine invokes a Simplex solver to satisfy a group of linear inequalities.

One problem that occurs in most logic programming language implementations is that once a solution is found, the path to the solution is forgotten; if one would like to solve several closely related problems, such as a mortgage payment problem with several different principal values, both systems would have to resolve the set of equations from scratch with the new values. This limitation makes most logic programming implementations undesirable for reactive systems, which perform small modifications on a large interconnected model. Also, while backtracking and unification are very powerful features of logic programming, they are both are quite costly, and may not be suitable for general-purpose programming. In addition, the behavior of backtracking may cause an environment written in a logic language to behave in an irregular manner due to unexpected delays [Freeman-Benson91].

Numerical Constraint Programming

Another major class of constraint systems are numerical, or algebraic, programming languages. Such languages allow the specification of simple algebraic relationships between objects. These relationships are typically restricted to the following dimensions:

- Noncyclic constraints. For example, `Z + Y = R + Z` is disallowed.
- Equalities and inequalities relating linear and near-linear expressions (non-linear expressions that can be reduced to linear), over the real numbers.

Numerical problems are typically solved using either local propagation or numeric methods such as Gaussian elimination, the Simplex algorithm, or relaxation [Borning79]. Local propagation is the most popular and fastest method, propagating known values through a constraint network to find unknown values. Spreadsheet programs are a typical example of a local propagation network. Unfortunately, local propagation is limited in the kinds of equations it can solve. More complicated constraints involving nonlinear equations can be solved using an iterative numerical process such as Newton-Raphson [Nelson85, Gleicher91]. Such processes are simple to implement, but they are often slow, and they have the additional disadvantage that they may not find the best solution with respect to a global objective function, but instead a locally-best one. To avoid this last problem, the programmer is often required to give a "hint" to the solver so that the solver may know where to start in order to find the globally-best solution. An important disadvantage to these approaches is that enough values must be known in the problem in order to propagate through the graph or to run through the iterative solver; these systems, being numerical solvers, cannot handle purely symbolic computations.

Symbolic Constraint Programming

A third class of constraint languages includes the symbolic constraint languages. Languages such as Maple [Char91], Macsyma [MathLab83], Mathematica [Wolfram88], and Bertrand [Leler88] solve equations via symbolic manipulation, rather than numerical processes. These systems often are based on a term rewriting substrate. Such systems transform equations using rules that maintain equivalence across rule application. These rules implement symbolic transformations such as Gaussian elimination to solve for variables representing numbers and expressions. While these languages often resort to numeric methods when the symbolic methods are not strong enough, the systems are still considered to be based on symbolic manipulation since they return symbolic results. For example, the set of equations

```
x + y = 15; x - y = r;
```

can be solved for `x` and `y` symbolically, yielding `x = (15 + r)/2, y = (15 - r)/2`. A major advantage of the symbolic approach is that incomplete information is not a problem. The system simply manipulates the equations that it has, and solves for the required values in terms of the existing expressions.

1.3 Object-Oriented Programming with Constraints

We use constraints to address the mentioned deficiencies in object-oriented languages. By adding the ability to specify constraints to an object-oriented language base, programmers can make explicit the implicit assumptions about internal state relationships. There are several advantages to this approach. First, by having such constraints in objects, a programmer reading the code can discover much more directly the intent of the object specification, where previously one would have to rely on comments or a thorough understanding of each method. Second, the system can generate methods automatically based on the constraints. Third, internal private methods that are often required for bookkeeping purposes (e.g., invoked to change the state in a particular way to maintain consistency) are no longer required, as the state changes satisfy the constraints directly.

We can use constraints to support correctness in inherited behavior as well. By requiring that constraints specified in both the ancestor object and the new object hold, the system deduces the behavior of the derived object strictly by the conjoined specification. Using constraints can even help discover inconsistent specifications that, in some cases, can cause a failed constraint during the satisfaction process.

Given these potential advantages, we would like to add the best features of constraint programming while maintaining the best features of an object-oriented viewpoint, encapsulation and inheritance. Note that this does not mean adding objects to a constraint programming system. We want to be able to think of our object in the same way, i.e., as encapsulated state-and-behavior, where we specify the internal state's consistency requirements explicitly via constraints.

Although there have been other languages that provided both constraints and objects [Borning79, VanderZanden88, Myers89, Avesani90, Freeman-Benson90, Wilk91], few have used constraints to define the fundamental object-oriented programming model. We would like instead to have constraints available as a fundamental language feature, integrated with the semantics, to be exploited at all levels of the system.

1.3.1 Messages vs. Constraints

The simplest approach would seem to be to allow both constraints and message-passing to coexist in the same system. However, merging constraints and objects is not a simple task. These two communication mechanisms, messages and constraints, are dramatically different. The object-oriented programming paradigm is a discrete state change paradigm: objects only change state when they are requested to by an external client through a specific, discrete event: a message being passed. The receipt of a message by an object causes time, for that object, to advance, changing its state. In short, messages between objects are discrete, local events.

The constraint programming paradigm, in contrast, is typically a continuous state change process (with some exceptions such as Kaleidoscope [Freeman-Benson90]): objects change state fluidly, in order to satisfy a larger global set of requirements, and in response to a concurrent set of "requests" on all of their attributes. Connections between objects are like wires between resistors, batteries, and capacitors on a circuit board. State changes in a constraint system, caused externally by some perturbation (setting a variable to a particular value, or adding a constraint to a global store), are propagated automatically throughout the system. Time advances when such a perturbation occurs. In short, constraints between such objects in a constraint system are continuous, global events.

	Temporal Scope	**Spatial Scope**
Messages	*Discrete*	*Local*
Constraints	*Continuous*	*Global*

Figure 1-4. Messages vs. Constraints.

Because of the continuous, global characteristics of constraints, the object-oriented feature of encapsulation does not exist in a pure constraint system. In such a system, there is no encapsulation boundary around an object; in particular, it would be difficult to characterize the set of events that occur to cause an object in a constraint system to attain a given state. This is because changes of state within a single "object" may cause arbitrary state changes throughout other objects in the system, depending on the the object's external constraints. Objects thus do not have any notion of discrete time, since state changes are not local to any object, but in fact global. However, the external event that initiates these state changes is discrete, and a sequence of external events does define the state changes of the constraint system as a whole.

In contrast, in a traditional object-oriented language, the sequence of messages sent to an object, along with the object's initial state, is sufficient to determine the object's new state (as long as the object has no shared state). Objects thus have a notion of discrete time that can advance when a message is received. In the case of a constraint system, the changes of variables by an external perturber determines the state of the system. This suggests that the encapsulation boundary of an object in a traditional object-oriented system, and the boundary between a constraint system and its external perturbers, play the same role, and that a message and a perturbation may be considered to be similar concepts.

Neither of these communication mechanisms is problematic; rather, each is useful for a particular way of thinking about systems, and both should be available in a programming system. What is needed is a model that accommodates both communication mechanisms. For objects that are considered to be single entities perturbed in a discrete way by external clients, the model could provide a message-based interface that allows the user to have control over a discrete state-changing process. These objects, in turn, can be defined in a part-whole hierarchy where they are related to each other solely via constraints; in such an object, each part subobject gives up its encapsulation for the seamless integration of continuously-constrained state.

1.3.2 The Problem

The problem that we are addressing in this thesis concerns the integration of the two communication mechanisms: message-passing between, and constraints on objects. How can these completely different communication mechanisms coexist in a single, uniform environment?

1.3.3 Solution Goals

If we begin with the fundamental ideas and structures of object-oriented programming and then add constraints, we must re-address the following issues:

- What is an object?
- What are messages?
- How are methods defined?
- How is encapsulation provided?
- How is inheritance handled?
- What can be constrained?

Specifically, we want to provide object-oriented programming where we can:

- Specify consistency requirements explicitly using constraint expressions in each object;
- Maintain encapsulation, so that each object is responsible for modifying its own state;
- Support encapsulation of inherited objects, to avoid having to know the implementation details of our ancestors; and
- Provide explicit control of the constraint satisfaction process, so that we can direct the solver to the solution that we want.

This thesis provides one approach to this problem.

1.4 Thesis Contributions

The goal of this thesis is to integrate constraints into an object-oriented programming model while retaining the benefits of the object-oriented paradigm. In reaching this goal, this thesis provides:

- A model of object-oriented programming with encapsulation and inheritance based on constraints;
- An execution model that uses term rewriting for symbolic constraint satisfaction in an object-oriented framework; and
- The design and implementation of a proof-of-concept language, Siri [Horn91], based on this model, using a single abstraction called a *constraint pattern*.

The model provides a framework for programming in an object-oriented style, while using constraints within objects to satisfy state-changes and to generate imperative method code given a declarative method specification. Explicit constraints make possible a specific encoding of the programmer's actual intent, rather than tedious details of how that intent must be carried out.

The execution model is an extension of Bertrand's *augmented term rewriting* mechanism. This extension, called *ATR+*, provides a general symbolic computation engine. An equation solver, based on Bertrand's, uses an equational normal form called an *ordered linear combination* [Derman84, Leler88] that can be easily manipulated and combined using term rewriting. The equation solver evaluates and solves constraints, and generates imperative code for changing the state of objects based on declarative methods. Objects defined with inheritance combine their definitions using constraint conjunction; the solution of the combined constraints helps to ensure semantic consistency, because constraints from both the object and its ancestor must be satisfied in the derived object.

The object-oriented language Siri implements these concepts using a single abstraction mechanism, the *constraint pattern*. The technical foundation of Siri is ATR+ with Bertrand's equation solving rules, combined with BETA's *pattern* abstraction mechanism. Siri can be considered as a generalization and extension of Bertrand that provides an interactive environment with changeable object state, using BETA's prefixing mechanism for inheritance, and patterns to provide a fully-hierarchical object structure.

1.5 Using the Constrained Objects Model with Siri

In this section we present first the rectangle example as coded using Siri. The following sections give an overview of Siri's execution model, and provide a brief discussion of Siri's design and implementation.

1.5.1 The Rectangle Example

Using constraints in Siri, we can can code the object `aRectangle` as described in section 1.1.1 in Siri by defining the relationships between attributes with constraints, and providing a single side-effecting method for changing the object's state:

```
aRectangle: {
   top, left, bottom, right: aNumber;

   width:   aNumber { self = right - left; self };
   height:  aNumber { self = bottom - top; self };
   center:  aPoint { x = left + width/2; y = top + height/2; self };

   "moveTo newCenter'aPoint": aMethod {
      width, height fixed;
      center = newCenter;
   };
};
```

Figure 1-5. `aRectangle` in Siri.

Each attribute is defined using constraints, constraining the attribute itself (`self`) to be equal to an expression involving other attributes. (Smalltalk-80 programmers will note that the use of `self` is different in Siri than in Smalltalk. Although the use of `self` in Siri methods does refer to the object enclosing the method, self in non-method attributes refers to the attribute in which the `self`-reference is used. Thus in the definition of `width`, `self` refers to `width`, rather than `aRectangle`, since we are naming the `width` from within its definition.) The `moveTo` method explicitly specifies that the `width` and `height` should not change and that the `center` should be moved to the new location. The programmer does not code the details of how these requirements

will be satisfied; instead, the consistency of the object's state is the direct responsibility of the system.

In Siri, the definition of `aRectangle` consists solely of nested constraint patterns and constraints. The single constraint pattern abstraction plays the roles of instance variables (`top`, `left`, `bottom`, `right`), non-state-modifying methods (`width`, `height`, `center`), state-modifying methods (`moveTo`), and objects (`aRectangle`) in a completely uniform way.

1.5.2 Evaluating MoveTo Using ATR+

When an instance of aRectangle receives the `moveTo` message, a constraint satisfaction engine computes the new consistent state of the object given the previous state and constraints on the object's `width` and `height`.

Specifically, at method instantiation time, Siri substitutes the values of attributes which are known to be fixed into an assertion equating the attribute with its value at instantiation time, and rewrites the attributes themselves to their residuals. Siri then evaluates the resultant expressions, binding attributes to new values. The method retains the residual expression for reevaluation at the next instantiation time. For example, Siri rewrites the residual for the method `moveTo` in `aRectangle`,

```
"moveTo newCenter'aPoint": aMethod {
   width, height fixed;
   center = newCenter;
};
```

to the following, using the constraint pattern that defines point equality and the definition for `center`:

```
width, height fixed;
center.x = newCenter.x; center.y = newCenter.y;
```

At instantiation time, Siri substitutes the `newCenter` into the residual and sets up the fixed constraints. If we assume that `width` is `100`, `height` is `50`, and `newCenter` is `(180, 230)`, the subject expression for evaluation becomes:

```
width = 100; height = 50;
center.x = 180; center.y = 230;
```

Siri then substitutes the residuals for `width` and `height` (shown in bold below),

```
right-left = 100; bottom-top = 50;
center.x = 180; center.y = 230;
```

and the bindings for the two `center` coordinates, which had been reduced from their original definitions to simpler ones involving only the `right` and `left` attributes (again, shown in bold):

```
right-left = 100; bottom-top = 50;
(right+left)/2 = 180; (bottom+top)/2 = 230;
```

This expression only contains the four attributes `top`, `left`, `bottom`, and `right`. Siri then solves for them and binds their new values using Gaussian elimination, as explained in section 4.2.3, completing the method evaluation.

1.5.3 Inheritance

We can use inheritance to code `aGoldenRectangle` from section 1.1.2 in Siri as follows:

```
aGoldenRectangle: aRectangle {
   "setWidth newWd'aNumber":  aMethod { center fixed; width = newWd; };
   "setHeight newHt'aNumber": aMethod { center fixed; height = newHt; };

   width = 1.618*height;
};
```

Figure 1-6. `aGoldenRectangle` in Siri.

In this example, the definition of `aGoldenRectangle` inherits attributes and constraints from `aRectangle`, while an additional constraint asserts the relationship between the `width` and `height` of `aGoldenRectangle`. The methods `setWidth` and `setHeight` explicitly describe the requirements that the `center` must remain the same, while changing the `width` and `height`. Note that we don't need to know or care whether `width` and `height` are stored state (like instance variables) or computed state (methods that compute a value) in the prototype `aRectangle`. Thus, we can change `aRectangle`'s implementation without affecting `aGoldenRectangle`.

1.6 Related Work

Other systems have been built to merge the object-oriented and constraint paradigms. These systems all differ from this thesis work in that they begin with a constraint system of some kind and then add objects to them. The object-oriented issues such as encapsulation and inheritance are either ignored or not completely addressed in these systems. In particular, objects are not fully encapsulated in that state changes within a single object can cause state changes to objects throughout the system. Ironically, the very first constraint systems, such as Sketchpad [Sutherland63] and ThingLab [Borning79], were grounded in an object-oriented framework, but did not provide the local control of object state that typifies object-oriented

programming.

The systems most closely related to Siri include Bertrand (section 1.6.1), ThingLab (section 1.6.2), and Kaleidoscope (section 1.6.3). The object-oriented language BETA, while not a constraint language, is also related to Siri; we discuss BETA briefly in section 1.6.4. Finally, we discuss other related systems in section 1.6.5, followed by a summary.

1.6.1 Bertrand

This thesis uses Bertrand [Leler88] as a foundation on which to build Siri. Bertrand uses augmented term rewriting for the solution of constraint problems by defining a symbolic equation solver with rewrite rules. Augmented term rewriting extends standard term rewriting by including a variable binding operation, facilities for defining new types and operators, and a name space that allows for the definition of structured objects.

Bertrand compiles a set of rules into a special finite state machine matcher that matches expressions in the constraint program and instantiates new expressions using the rewrite rules. Bertrand runs a program by applying the compiled matcher/instantiator to a set of expressions that are rewritten until no matches are possible, returning the final expression. The solution is expressed in terms of values for variables, where values may be symbolic expressions.

There are many features provided by Bertrand that make it an excellent base from which to build an object-oriented constraint language: its matching process is fast, and its speed is independent of the number of rules being matched; it is extensible and flexible; it supports conditional constraints; it is computationally complete; equation solving is an appealing tool for simplifying, generating, and optimizing code, since it can handle expressions with partial information; and term matching and the type system support evaluation of polymorphic expressions. Unfortunately, Bertrand has several limitations that make it unsuitable for general purpose programming:

- Bertrand is a batch system, in which objects exist only during the evaluation process.
- Objects may not be modified at runtime, since Bertrand provides only single-assignment semantics. There is thus no message passing or methods on objects.
- Bertrand does not allow block-structured programs; all Bertrand programs are written at the same lexical level.
- Bertrand does not provide inheritance of any kind.

While Bertrand has many desirable features, it falls short of being a true object-oriented language. Siri resolves Bertrand's limitations by providing an interactive, incremental programming model with the object-oriented features of encapsulation, message passing, and

inheritance in a block-structured framework.

1.6.2 ThingLab and ThingLab II

ThingLab [Borning79], a constraint-oriented, direct-manipulation simulation laboratory built on top of Smalltalk [Goldberg83], provides a language and a graphical interface for describing collections of related objects. Objects in ThingLab are related to each other via constraints on subparts; modification of an object generates a satisfaction method that uses local propagation to update dependent objects. In the case that local propagation fails, ThingLab resorts to a numeric method, relaxation, to generate a solution. ThingLab's constraints are multidirectional, but the programmer must define satisfying methods for each direction manually.

ThingLab cannot simplify expressions symbolically, or solve for unknowns when incomplete information is presented. Though ThingLab does compile satisfaction methods, they do not take advantage of potential equation simplifications, since the compiler has no global knowledge of the equations being compiled. Algebraic reductions with constraint patterns have the potential to generate much simpler and faster methods.

ThingLab II generalized ThingLab to allow a hierarchy of constraint strengths and to allow specification of default or desired behavior in underspecified cases. A new algorithm, called DeltaBlue [Freeman-Benson89], made it possible to quickly add or remove constraints with different strengths to the appropriate parts of the constraint graph. ThingLab II was intended for user interface construction, and as such provided a new primary entity, a "thing," which was an object in the Smalltalk sense, plus constraints. The user can define constraints via Smalltalk, or graphically in some cases. Primitive constraint satisfaction methods, for base cases such as `a*x = b`, are defined in Smalltalk, though the user can add more sophisticated methods.

While ThingLab and ThingLab II do have objects, they do not use a message interface for communication. Both systems are constraint oriented, and use external perturbation directly for causing changes within the constraint network. In short, the ThingLab and ThingLab II models do not support encapsulation; thus, there is no local control of state and no explicit interfaces for object modification.

1.6.3 Kaleidoscope

Kaleidoscope [Freeman-Benson90] is the system most closely related to the model presented in this thesis. Kaleidoscope is a mixed imperative and declarative language, and, like ThingLab II, has constraints arranged in a hierarchy of strengths. The system includes a special mechanism called a constraint constructor, which is similar to a rewrite rule; the constraint constructor decomposes, or splits, complex constraints between objects into simpler constraints on their subparts. Kaleidoscope does not perform equation solving, but uses the satisfaction algorithms pioneered in ThingLab and its successors.

Kaleidoscope showed that constraints and imperative programming can be merged in an elegant and general language. Kaleidoscope allows the programmer to mix freely variable assignments with constraints that are to be maintained by the system. As in imperative systems, assignment is the state-changing mechanism; the updating of dependent objects occurs automatically. Kaleidoscope considers variables to be sequences of values through time; thus assignment to a variable is simply a constraint that relates the variable at a time `i+1` to the variable at time `i`. Adding `6` to the variable `x` is understood as the constraint over time: `x[i+1] = x[i] + 6`. The programmer may also restrict constraints to a range of times, adding them at the start time and removing at the end time. This is especially convenient for coding user interfaces, as in "`while the mouse is down, the following constraints hold: ...`"

Since time can be advanced at any point in the program, and constraints may be added and removed easily, Kaleidoscope is a more general solution for mixing constraints and imperative code than the constrained objects model. However, the size of the constraint network can become an efficiency issue when creating very large systems. We can consider our model to be a restriction of Kaleidoscope in order to streamline it for greater simplicity and potential efficiency, as well as to provide a stronger object-oriented focus with support for inheritance, encapsulation, and message passing.

Although Kaleidoscope's and Siri's goals are similar, Siri addresses the constraints-and-objects problem quite differently. Siri's objects are strongly encapsulated; clients may modify objects only via their message interfaces, while in Kaleidoscope, objects change their state due to constraints on their subparts. Programmers write methods in Siri explicitly for perturbation of objects, with constraints to hold certain attributes constant while others are allowed to change. In contrast, Kaleidoscope controls the satisfaction process via a hierarchy of constraint strengths, which is more flexible but also less direct. Also, Siri provides some important features that Kaleidoscope does not: equation solving semantics for symbolic solution of equations and compilation of satisfaction methods; a uniform mechanism for multiple inheritance; and a single abstraction mechanism to describe all types of structures in the system. Finally, Siri's design is simpler and potentially easier to understand.

1.6.4 BETA

BETA [Kristensen89] is an object-oriented language based on a single abstraction mechanism called a *pattern*, with block structuring and prefixing for inheritance. In this thesis we generalize BETA's imperative patterns to Siri's constraint patterns: block structured organizations of constraint expressions.

BETA's *prefixing*, a generalized inheritance mechanism, provides a variety of interesting variants on inheritance including inheritance on methods rather than just classes. Siri uses prefixes as well to allow the inheritance of constraint expressions as well as object attributes.

Siri's constraint patterns can be considered a constraint-based relative of the BETA pattern. While BETA's patterns specify imperative actions, constraint patterns declare constraint expressions, merging the benefits of BETA's hierarchical structure into an object-oriented

constraint system. Siri's constraint patterns further distinguish themselves from BETA patterns by providing a multiple inheritance capability.

1.6.5 Other Related Languages

Part of the Garnet system [Myers89] is a unidirectional constraint language called KR built on Common Lisp. KR provides unidirectional constraints using value propagation, and as such cannot handle symbolic constraints or simultaneous equations. However, KR and Garnet are quite efficient in this restricted domain, and have been used for some very large projects. Garnet has been extended to use ThingLab II's hierarchical constraints with the DeltaBlue algorithm in a system called Multi-Garnet [Sannella92].

CONSTRAINT [VanderZanden88] is a language designed with interactive graphical applications in mind. Graphical objects and subobjects may be displayed and modified, while satisfying a set of constraint equations, ordering these equations for optimal update to ensure real-time interaction. The user can display and modify graphical objects and subobjects, while the system maintains the consistency of a set of constraint equations. For efficiency, CONSTRAINT used an algorithm similar to DeltaBlue for solving equations. Because of this, it was limited to noncircular, multidirectional constraints; this made it unable to solve simultaneous equations. In addition, CONSTRAINT does not provide a general model for object-oriented programming, as does Siri; it is restricted to the domain of interactive graphical applications.

Equate [Wilk91] is an object-oriented constraint solver that uses rewrite rules. Equate provides objects with internal constraints, but does not provide messages to objects; instead, it devises a plan using backtracking search, based on declarative requirements, that specifies perturbations of objects. Its use of backtracking conflicts with the desire to have strict object-oriented behavior; in the constrained-object model, the programmer directs the resatisfaction process to a unique solution by fixing object values. In addition, object encapsulation is preserved during the operation of the solver; [Freeman-Benson91] points out this is not necessary, and in fact disallows the system from making some optimizations that do not affect the user's view of the model.

1.7 Roadmap to Thesis

Chapters 2, 3, and 4 address each of the main thesis contributions: the *constrained objects* model; the programming language Siri; and the execution model. In Chapter 2, we present constrained objects, a semantic model of object-oriented programming based on constraints. Chapter 3 discusses Siri, a programming language whose main feature is its compact size, and its single abstraction mechanism, the *constraint pattern*. Chapter 4 presents Siri's evaluation mechanism and implementation details, as well as the constraint satisfaction engine, object structures, and runtime kernel. Chapter 5 discusses optimization strategies for improving Siri's execution performance and compactness, and Chapter 6 provides a summary, conclusions, and suggests an outline for future work. Related work is distributed across the chapters, including

this one, as appropriate.

The appendices present a variety of Siri code examples. Appendix A provides some larger Siri program examples: a stack; the factorial function coded in various styles; an electrical component simulation with sample circuits; a graphical scroll bar; a rational number package; and an outline of an experimental user interface based on documents floating in three dimensions called Hyperfax. Finally, Appendix B is a listing of the intrinsic Siri code for the runtime system.

Chapter 2
The Model

This thesis incorporates constraint satisfaction into a model of object-oriented programming. One of the fundamental features of object-oriented programming is encapsulation: a client object may not arbitrarily change the internal state of another object. Instead, a client sends a message requesting a change; the object handles this request using methods within the object that are coded to perform the particular request and have direct access to the object's state. By doing this, object-oriented programming provides two important features: first, the intent of a request is separated from the actual implementation, promoting modularity; and second, the consistency of the object's internal state is maintained, since arbitrary changes are disallowed.

Constraint languages, on the other hand, typically do not provide modularity in this way. While constraint languages can provide objects, the very mechanism that propagates changes between objects, that is, constraints, breaks the modularity. Objects no longer are fully responsible for their own internal consistency, since any constraint outside of the object that constrains part of its state can cause it to change.

In order to give an object full control over its own state, the model presented here provides two levels of communication. Between objects, communication is performed strictly via message-passing; there is no mechanism for directly changing an object's internal state from a client. Within an encapsulated object, however, all communication among internal state subobjects, and their subobjects, is accomplished via constraint expressions only; they may not send messages to each other. In Figure 2.1, boxes represent encapsulated objects with subobjects as rounded-corner rectangles; lines represent constraints among subobjects, and arrows indicate messages sent between encapsulated objects.

Message passing between objects defines a more precise communication interface with protection of the internal implementation. Constraints within an object are used to maintain consistency of the state, and to make it simpler to define methods that change it. The two-level organization in this model provides both communication mechanisms: messages for global communication, and constraints for local. Note that messages and constraints are not equal partners: constraints perform a strict and limited role in maintaining an object's state consistency under perturbation, while messages only are used for communication between objects.

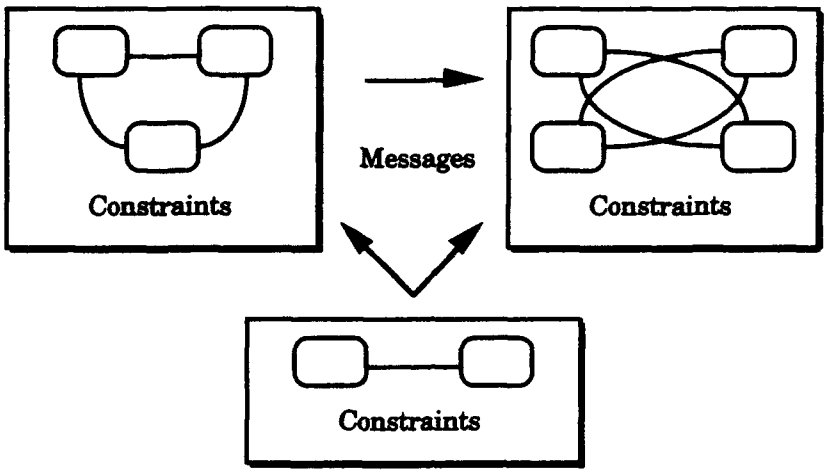


Figure 2-1. Messages and constraints in encapsulated objects.

A major implication of the two-level organization is that the programmer is restricted from using constraints to communicate between encapsulated objects. If an object's state is dependent upon another object, and the two are not parts of a single encapsulated object, then they must use messages for communication. As in traditional object-oriented programming, a dependent object explicitly handles changes in state of external objects via notification messages. The notified object then accesses their state values, again with messages, and incorporates these values into its own state.

Constraints within objects make possible a useful extension to object-oriented programming. If a message is sent to an object, it may be that the particular message with its arguments cannot be accepted by the object without violating one or more constraints. These constraints can be on the object as a whole, or on the parameters of the message (such as pre- and post-conditions). In such a case, an exception can be raised and passed back to the client.

2.1 Object Interface: The Outside View

Clients view an object from the outside by its interface. The interface to an object consists of a set of *attributes* that define their external characteristics, and *methods* that perform operations to change the object's attributes in a particular way. Attribute values are accessed, and methods are invoked by sending *messages* to the object.

The graphical rectangle example (Figure 1-1) from section 1.5.1 provides the following interface:

```
aRectangle: {
   -- Attribute interface
   top, left, bottom, right:  aNumber;
   width, height:             aNumber;
   center:                    aPoint;

   -- Message interface
   "moveTo newCenter'aPoint": aMethod;
};
```

Figure 2-2. `aRectangle`**'s interface.**

Thus, a client of `aRectangle` can send a message requesting the rectangle's `width` attribute and receive `aNumber` in response, or modify the rectangle's position by sending it a `moveTo` message with a new `center` point.

Similarly, the interface to a temperature object that provides Celsius and Fahrenheit attributes may be given as:

```
aCFTemp: {
   C, F: aNumber;
};
```

Figure 2-3. `aCFTemp`**'s interface.**

In this case, there is only an interface to the two attributes of `aCFTemp`, with no interface for changing the temperature value.

2.1.1 Attributes

A client accesses information about an object by sending a message that requests an attribute value. In the interface, an attribute is the name and signature of an operation[1] that returns a value representing some property of an object. For example, the rectangle's attributes include its `top`, `left`, `bottom`, and `right` coordinates; its `width` and `height`; and its `center` point. Attributes are a way to talk about an object's abstract characteristics. Object attributes may not be side-effected, assigned, or constrained from the outside; they are read-only.

The type of an object's attribute is typically not the same as the type of the object. However, in some circumstances, such as in a recursive data type, it may be a pointer to an attribute of the same type, such as the `nextLink` attribute in a `LinkedList` object.

[1]The signature of an operation is a list of the types of its parameters and its result.

2.1.2 Methods

A client changes another object's state by sending a message that invokes a *method*. A method is the name and signature of an operation that may change an object's state. Invoking methods is the only mechanism available for a client to modify an object; method invocation evaluates object-specific code to change the object's state. Methods are a way to decouples the intent of the user from the details of how it is to be carried out. For example, the above rectangle example includes a method definition for moving the rectangle by its `center` point.

Separation of non-side-effecting attributes from side-effecting methods allows strict control over the state modification process. This separation makes it possible to talk about keeping certain attributes fixed while changing others.

2.2 Object Implementation: The Inside View

A constrained object is implemented using three different kinds of internal components: *attribute* definitions that collectively define the state of the object; *constraints* that limit potential states of the object to a subset of valid states; and *methods*, internal message definitions that define ways to perturb the object's state.

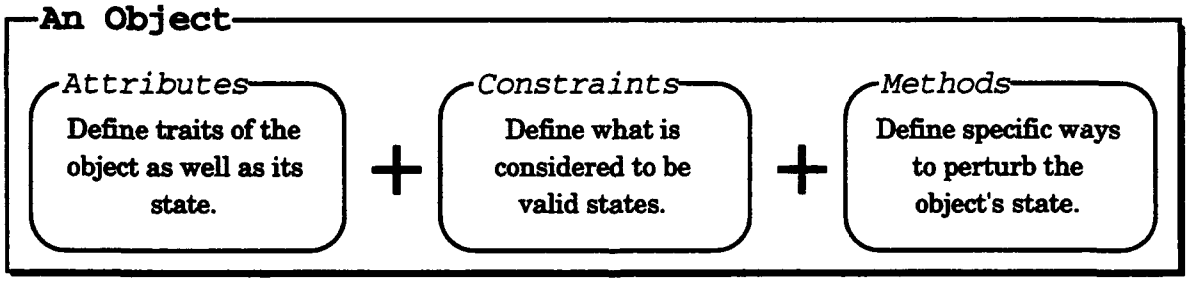


Figure 2-4. The parts of an object.

2.2.1 Attributes

Attributes define the characteristics of the object as well as its internal state. Attributes may be externally *visible* or *hidden*; these may be independently *owned* or *shared*.

Visible attributes are exactly those that allow external clients to view the object's characteristics. These provide information to clients about the internal state of the object without providing direct access.

Hidden attributes are those defined for internal use only. The distinction is available for the programmer's convenience; hidden attributes are typically attributes that are useful in an object's implementation, but inappropriate in the interface.

Owned attributes define the state of an object. The object itself is fully responsible for their values, and thus owned attributes are not shared externally with any other object. Some attributes can be considered as stored state, similar to instance variables in Smalltalk classes [Goldberg83]. Other attributes can be defined as functions on other attributes; these are called *virtual parts* [Freeman-Benson91]. This distinction between computed state and stored state is in fact unnecessary; the programmer need not decide what is to be computed and what is to be stored. The union of all owned attributes is the state of the object; optimizations to store certain parts directly and to compute others can be left to the underlying system.

Shared attributes can be thought of as references or pointers to common external objects. Shared attributes provide a mechanism for knowing about, and passing messages to, such objects. Though shared attributes to external objects do not count as part of an object's state directly, an object's state can still be a function of attributes of external shared objects as well as owned ones.

A shared attribute specifies the type of object to which the attribute will refer, but not the exact object itself. After the enclosing object has initialized its state, it can set its shared attributes, typically by receiving the object to be shared as a parameter in a message. For example, we might add a shared attribute to `aRectangle` to point to the window manager that handles its drawing:

```
aRectangle: {
   top, left, bottom, right: aNumber;
   desktop: ^aWindowMgr;
   ...
};
```

Then, two rectangles may both share the same window manager for their `desktop` attributes (figure 2-5).

When the shared window manager object changes its state, the underlying system will notify the two rectangles, which may then change their own state as well, by retrieving attribute values via the window manager's interface.

Most constraint-object programming in our model will not use references, however. In a standard part-whole hierarchy of an object that consists solely of owned attributes, constraints cannot occur between attributes that cross the main object boundary. In this way, a firewall is maintained between the object and its clients such that the constraint satisfaction mechanism can be used within the object for satisfaction, and the message-passing mechanism can be used between high-level objects for communication.

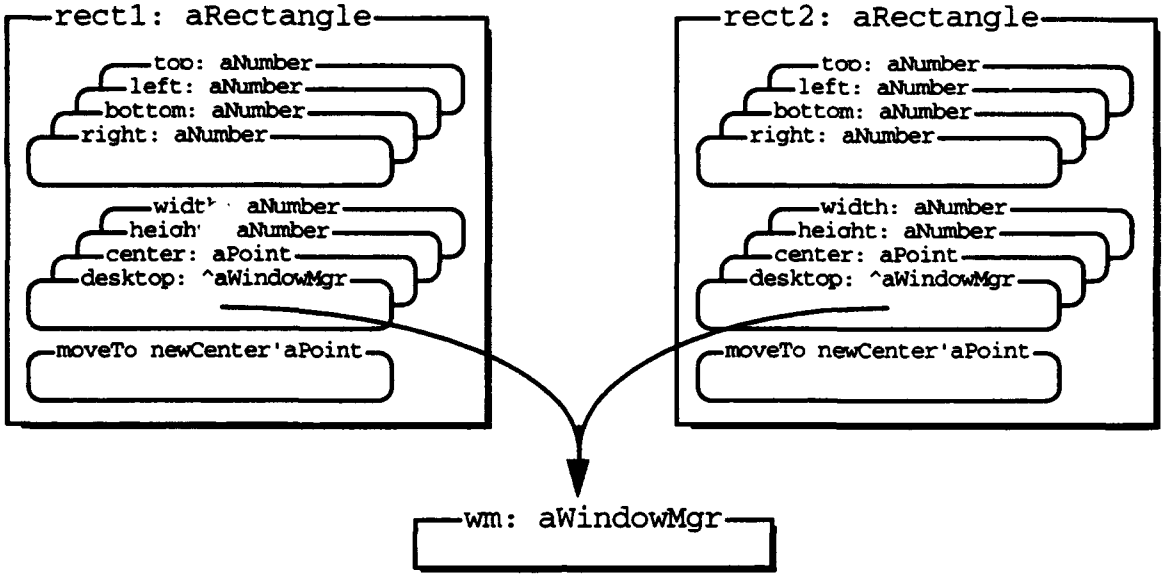


Figure 2-5. The `desktop` shared attribute.

All combinations of visible, hidden, owned and shared attributes are possible:

- Visible owned attributes are views on the internal object state that are of interest to clients.
- Visible shared attributes are used for communication with other objects that are also of interest to clients.
- Hidden owned attributes are used for internal computation; they are hidden because they do not provide a useful external view of the object.
- Hidden shared attributes are used for communication with other objects that, for protection purposes, should not be visible to clients.

Examples of the use of each kind of attribute combination will be given in sections 2.2.2, 2.2.3, and 2.3.2.

2.2.2 Constraints

An object's attributes are related to each other via constraints. Constraints are expressions that specify a particular relationship to be maintained between the attributes, such as an equality relation, an arithmetic relation, or a user-defined relation. All constraint expressions within an object are conjoined, requiring them all to be satisfied simultaneously.

Constraints within an object provide communication paths between the attributes. Constraints thus replace local message-passing to internal objects.

Some attributes encode pure state, and so their values can be returned directly to the client; others must be computed as a function of the other attributes of the object. The actual functions defining such attributes are implicitly given using constraints; the underlying system uses the constraint satisfaction mechanism, described later, to build *satisfaction methods* for each attribute. These satisfaction methods are invoked by the system to compute attribute values when required.

Valid states of an object are determined by constraints only on owned attributes, both hidden and visible. Constraints are not allowed on shared attributes, as this would violate the modularity of the object: it would be possible to modify the object without going through the message interface, simply by changing a shared object's state, causing aliasing.

The following is a more complete implementation of `aRectangle` that includes two attributes in addition to the shared desktop: the `area`, and the `aspectRatio`. In addition, methods are provided for changing the `center` point and the `area`. This example highlights the different kinds of attributes and constraints that define the object.

```
aRectangle: {
   top, left, bottom, right: aNumber;   -- Owned, visible
   desktop: ^aWindowMgr;                -- Shared visible

   width:          aNumber  { self = right - left;   self }; -- Each of these
   height:         aNumber  { self = bottom - top;   self }; -- attributes is
   area:           aNumber  { self = width * height; self }; -- both owned and
   aspectRatio:    aNumber  { self = width / height; self }; -- visible
   center:         aPoint   { x = left + width/2; y = top + height/2; self };

   "moveTo newCenter'aPoint": aMethod { ... };      -- Owned, visible method
   "setArea newArea'aNumber": aMethod { ... };      -- Owned, visible method
};
```

Figure 2-6. An extended version of `aRectangle`.

`aRectangle` in this case uses only visible attributes. All attributes except `desktop` are owned attributes; `desktop` is a shared attribute, pointing to the window manager that displays the rectangle.

Note how constraints are used to define the `width` and `height` of `aRectangle` in terms of equations relating `self` to the `right` and `left` coordinates for `width`, and the `bottom` and `top`

coordinates for height. Also, constraints are used to define the x and y attributes of the center point of aRectangle; these constraints specify a given relation between the attributes of the center point (x and y) and attributes of the rectangle itself (top, left, width, and height).

To provide an implementation for the temperature object aCFTemp (section 2.1) we simply add the single constraint that relates the two attributes. The definition is as follows:

```
aCFTemp: {
   --Visible owned attributes
   C, F: aNumber;

   --Constraints on the attributes
   F - 32 = 9/5 * C;
};
```

Figure 2-7. A definition of aCFTemp.

2.2.3 Methods

Methods are the only mechanism for changing state within an object. Because attributes are typically related to each other via constraints, changing one will often change others to maintain internal state consistency. Methods are the mechanism for specifying explicitly what attributes will change, and how the others should change in response. aRectangle provides methods for moving its center point and changing its area. In both of these methods, we need to specify which other attributes should remain unchanged while updating the center point and the area to their new values.

Normally, methods consist of imperative code in traditional object-oriented programming. In this model, methods consist of four different kinds of statements:

- Additional declared constraints that are invariant during an invocation of the method;
- Operations for holding attributes fixed during the constraint satisfaction process;
- Assignments to attributes to set their values; and
- Messages to objects referred to by shared attributes.

These statements specify how to modify the object's state when a message is received. When the corresponding method is invoked, the object's constraints are conjoined with the method's constraints, and the constraints are then solved. The solution of the constraints yields new values for attributes, and thus the new state of the object. The object's state is changed atomically: no client may observe an object in an inconsistent state.

The object's possible degrees of freedom are controlled in two ways: by fixing and assigning attributes. Fixing an attribute to its current value or assigning an attribute to a new value reduces the degrees of freedom of the system. If enough degrees of freedom have been specified, the system can compute the object's new state by solving for the remaining attributes in terms of the ones given.

For example, we code the method that moves the rectangle while keeping the `width` and `height` constant as shown in section 1.5.1:

```
"moveTo newCenter'aPoint": aMethod {
   width, height fixed;          -- Fixing attributes that don't change
   center = newCenter;           -- Assigning the new center.
};
```

In the `moveTo` message, three equations are added temporarily to the set of constraint equations. Constraints equate the `width` and `height` attributes to their old values, and the `center` attribute to its new value. These equations provide enough additional information to solve uniquely for the remaining attributes, the `top`, `left`, `bottom`, and `right`. The result is that the method changes the state of the rectangle in such a way that the `width` and `height` do not change. Since no other attributes are mentioned, it is assumed that they are free to change in order to accommodate the assignment of the `center` point to a new value.

Compare this example code in the above example to the C++ code in section 1.1.1 that directly modifies the state variables of the object. The explicit mention of method invariants in a constrained object is a significant improvement over method definitions in traditional object-oriented programming. By encoding methods in this way, the programmer directly specifies the full intent of the method's effect, including what is to remain fixed and what will be changed. If this were a standard object-oriented language method, this information would not be available at all in the method or the object definition, but would be implicit in the code written; it would be up to the programmer to interpret the method and discover the underlying model.

Similarly, we code the method that changes the rectangle's area by fixing the `aspectRatio` and the `center` point:

```
"setArea newArea'aNumber": aMethod {
   aspectRatio, center fixed;          -- Fixing attributes that don't change
   area = newArea;                     -- Assigning the new area.
};
```

2.3 Communication Mechanisms

A system based on this model consists of a set of objects that communicate with each other, retrieving information and initiating actions that change the state of the system. The two communication mechanisms available in the model are *message passing* and *constraint evaluation*.

Message passing is the mechanism for requesting the value of an object attribute, or for initiating the change of an object's internal state between two encapsulated objects. These two cases of message passing provide the read-only and modifying operations on objects, respectively.

Sending a message requesting the value of an attribute provides a client with a particular view on an object. The system passes the attribute's value back to the client without changing the object's state.

An object may also change its state when it receives a message. Modification messages may come from two sources: the system itself, and other encapsulated objects defined by the programmer.

2.3.1 System Messages

The underlying system sends messages to communicate special kinds of events. There are three system messages that an object may receive in its lifetime: `initially`, `changed`, and `finally`. These three system-invoked messages cover all of the potential state changes within a system: creation of an object, changing of state during the object's lifetime, and disposal of an object, and thus provide an interface to the programmer for noticing and acting upon these events.

The `initially` *Message*

When an object is first created, the system sends an `initially` message to it. Objects use this message to initialize their own state without introducing permanent constraints.

The `changed` *Message*

Because a given object may be dependent on information shared among many different objects in the system, there must be a way for that object to access the information and to update its own state when necessary. Since communication between objects is done solely through messages, the system sends `changed` messages for notification to all objects with shared state. Whenever an object's shared attribute changes state, the system sends the object a `changed` message.

Usually an object will not need to notice that its own state has changed, unless there is an external client (such as a display screen) that also needs to be notified. Typically, however, these external clients will have references to the affected object, and will be notified directly through their own `changed` message interface.

Objects have full control over how they handle changed messages for their shared attributes. In particular, objects are not forced to recompute their state by incorporating the changes. They may simply note that the particular shared attribute had changed, and incorporate the change at a more convenient time. For example, during recomputation of a page layout in a word processing system, many objects will be changed. Instead of incorporating the changes incrementally, as they occur, the screen object can collect the changes and update them all at once to minimize screen flicker.

Handling changed messages is similar to providing cross-object constraints. However, we are limited to handling dependency relationships; changed messages cannot be used to specify and satisfy simultaneous equations, for example. Dependencies with loops cycle until the objects in the dependency chain quiesce to a stable state.

The `finally` *Message*

When an object is about to cease to exist, the system sends the `finally` message to it. A typical use of the `finally` message would be to detect when a screen object is no longer being referenced and to remove it from the display. This is similar to the concept of finalization in garbage collection [Harms89], and is subject to the same ordering concerns that are important in finalization. For example, an object cannot receive a `finally` message until all other objects that are dependent on it have evaluated their `finally` messages, and are no longer dependent on the object's state.

2.3.2 A System Message Example

We can use these system messages to communicate between separate objects that work together in an application. For example, imagine that we would like to display the outdoor temperature in degrees Kelvin, Celsius, and Fahrenheit simultaneously. We have an instance of `aTempSensor`, the system's `outdoorTempSensor`, that detects the temperature, and whose single attribute is the temperature in degrees Kelvin.

```
outdoorTempSensor: aTempSensor {
   value: aNumber { --constrained by a primitive sensor reader-- };
};
```

Using the sensor, we can define an object called `theOutdoorTemp`, similar to `aCFTemp`, that provides a `display` string as a visible attribute.

The object initially fetches the current temperature value, and sets the shared attribute `sensor` to the `OutdoorTempSensor`. To get the outdoor temperature, we maintain a hidden shared attribute that references the sensor, notes when it changes, and updates the object's state, including the display string, when necessary. For the display string to display the three different temperature units, `theOutdoorTemp` maintains hidden owned attributes `K`, `C`, and `F` for converting between the various temperature values. Finally, when the object ceases to exist, the system beeps, notifying the user that the object has been reclaimed (for debugging purposes, say).

Given these requirements we define the object `theOutdoorTemp` as follows, with hidden attributes marked with a hash (#):

```
theOutdoorTemp: {
   --Visible owned attributes
   display: aString;

   --Hidden owned attributes
   #K, #C, #F: aNumber;

   --Hidden shared attribute
   #sensor: ^anOutdoorSensor;

   --Hidden interface for system-invoked messages
   #"initially":              aMethod { sensor is OutdoorTempSensor;
                                        K = sensor.value; };
   #"sensor.value changed":   aMethod { K = sensor.value; };
   #"finally":                aMethod { Beep ! };

   --Constraints on attributes
   display = "K=" + K + " C=" + C + " F=" + F;
   K - 273 = C;
   F - 32 = 9/5 * C;
};
```

Figure 2-8. `theOutdoorTemp`.

Since both `C` and `F` can be computed given `K`, the `"sensor.value changed"` method provides enough information so that all three attributes of the `OutdoorTemp` are updated whenever the `sensor's` value changes.

2.3.3 User Messages

User messages are sent between encapsulated objects from within method code. The general form of sending a message is

```
receiver.messageExpr
```

where `receiver` is the object receiving the message, and `messageExpr` is a message expression to evaluate in the receiver's context. A message expression can be the name of an attribute to evaluate or a method to invoke. The expression

```
myRectangle.width
```

sends a message to `myRectangle` that returns with the value of the rectangle's `width` attribute.

To change an object's state, we send it a message expression, along with typed parameters that are specified in the message name. For example, sending a message to move a rectangle called `R` to a new location `p` can be accomplished as follows:

```
R.(moveTo p);
```

This expression sends the message `moveTo p` to the object `R`. The object `R` evaluates the message internally by setting the environment to `R` and then evaluating the expression in that context.

2.3.4 Constraint Evaluation

Constraint evaluation occurs within an object when a message is received. The process is based on solving algebraically for certain attributes in terms of the others using a variant of augmented term rewriting [Leler88].

The set of constraints being solved include the constraints defined in the object itself, as well as related constraints defined in outer scopes. If the constraints given are underspecified and there is no unique solution, the solution will be returned in symbolic form. In the case that a set of values must be returned, for example to display graphics, it is up to the programmer to make sure that appropriate default values are initialized.

For example, in `aRectangle`, we initialize its attributes using an `initially` message:

```
aRectangle {
   initially: aMethod {
      width = 100; height = 50; center = point (400, 300);  -- Constraints
   };
   ...
};
```

The constraint satisfaction engine solves for the `top`, `left`, `bottom`, and `right` coordinates given the `width`, `height`, and `center` point. Using algebraic manipulations, the method determines the values for the free attributes and assigns the `top` to `275`, the `bottom` to `325`, the `left` to `350`, and the `right` to `450`, the only values that satisfy the constraints given in the method.

Similarly, to evaluate a method with fixed and assigned attributes, the system combines constraint expressions within the object with the method code to form a complete specification for the state change. The method computes the new state of the object in three steps: first, equality constraints replace assignments by equating attributes with their new values; second, equality constraints replace fixing operations by equating attributes with their existing values; and finally, the system solves for the remaining free attributes.

For example, to evaluate a rectangle's `moveTo` method, where the rectangle's `width` and `height` are `200` and `100` respectively, and the argument `newCenter` is the point `(300, 240)`, the system would solve the constraints

```
width = 200; height = 100; center = point (300, 240);
```

and would then perform the same constraint satisfaction process as explained in the `initially` example to determine values for the attributes `top`, `left`, `bottom`, and `right`.

2.4 Object Instantiation

An object is created by making an instance of an existing object, its *prototype*, from which it obtains its essential characteristics: its attribute and constraint definitions. Evaluation of the declaration creates a subobject that plays the role of an object attribute. Thus, the expression `n: aNumber;` as defined in

```
R: {
   n: aNumber;
   ...
};
```

creates an instance of `aNumber` that becomes an attribute of `R`; this instance is named `n` in the environment defined by `R`.

When a new object is created from a prototype, the following conditions hold:

- For each owned attribute in the prototype, there is a corresponding owned attribute in the new object, with its definition being a copy of the prototype's owned attribute definition;

- For each shared attribute in the prototype, there is a corresponding shared attribute in the new object, with a reference to the same type of object.

- For each method definition in the prototype, there is a corresponding method in the new object, with its definition being a copy of the prototype's method definition; and

- The new object's constraints are copies of the prototype's constraints.

Instantiation of a prototype thus results in a new object with a copy of the prototype's specification. The values of the new object's attributes are independent of the values of the old object's attributes; only the specification is copied.

To illustrate instantiation, we present several small examples where we can create a point in different ways. The following declaration creates a point with uninitialized state; it is expected to be constrained at some future time:

```
p: aPoint;
```

To create a point whose state is always constrained such that its `x` coordinate is `15` and its `y` coordinate is `20`, we can write the following:

```
p: aPoint { x = 15; y = 20; };
```

Since the point is constrained to be always at that location, the state of the object `p` will never change: `p` is the constrained point `(15, 20)`. Finally, we can create a point whose initial state is given, but that is otherwise unconstrained:

```
p: aPoint {
   initially: aMethod { x = 30; y = 50; };
};
```

This creates a point with initial `x` and `y` coordinates of `30` and `50` respectively. `initially` message sets the values of the point attributes, constraining them only during method evaluation. The object `p`'s state can change in the future as methods are invoked.

2.5 Inheritance

The implementation of an object may inherit from any number of other objects; thus the model provides multiple inheritance. The objects being inherited are known as *prefixes*, and the object being extended is called the *base*. Prefixes are objects that provide a set of initial attributes and constraints. By merging each prefix to the base in order we achieve a *derived* object whose attributes and constraints are defined by all of the participating prefixes as well as the base.

We define the merge operation on objects as:

- the conjunction of the constraints of each object; and
- the union of the sets of attributes of each object.

Prefixing conjoins the constraints of the prefix with the base, and merges together the attributes of the prefix and the base to produce the derived object. Since the attributes of the objects being combined may have the same names, we define also a conflict resolution procedure: any attributes in the prefixes and the base with common names are recursively merged to create a single attribute in the derived object. We use the `inner` construct (generalized to alternation; see [Thomsen86]) to enforce an ordering that is important when merging imperative message passing sequences.

2.5.1 Attribute Inheritance

Prefixes provide a set of initial attributes that are unioned together with the base. If an attribute is provided by more than one source, all of the namesakes in turn are unioned together to create a single derived attribute.

More specifically, consider the object definitions `P` and `Q`, where `Q` uses `P` as a prefix for inheritance. Both `P` and `Q` define an attribute `x`:

```
P: {
   "P expressions..."
   x: { "P.x expressions..." };
};

Q: P {
   "Q expressions..."
   x: { "Q.x expressions..." };
};
```

Since both `P` and `Q` define an attribute `x`, their definitions are merged by prefixing `P`'s `x` to `Q`'s `x`, as if we had written

```
Q: P {
   "Q expressions..."
   Q.x: P.x { "Q.x expressions..." };
};
```

The `Q` that results is thus

```
Q: {
   "P expressions"
   "Q expressions"
   x: { "P.x expressions..." "Q.x expressions..." };
};
```

In general, if an object definition Q is prefixed by P1, P2, ... Pn, then the following holds:

- If Q defines an attribute x, and it is not defined by any of P1, P2, ... Pn, then its definition stands. The result is the definition `Q.x: {"Q.x expressions"};`

- If Q defines an attribute x, and any of P1, P2, ... Pn also define x, then Q.x is prefixed by all of the attributes in P1, P2, ... Pn that are named x. The attributes P1.x, P2.x, ... Pn.x are prefixed to Q.x in the same order as the P's are prefixed to Q. The result is the definition `Q.x: P1.x, P2.x, ... Pn.x {"Q.x expressions"};`

- For any attributes not defined by Q but defined in any of P1, P2, ... Pn, then each such attribute x will be defined in Q as an empty body prefixed by all of the attributes in P1, P2, ... Pn that are named x. The attributes P1.x, P2.x, ... Pn.x are prefixed to the empty body in the same order as the P's are prefixed to Q. The result is the definition `Q.x: P1.x, P2.x, ... Pn.x { };`

By combining objects with prefixing in this way, specialization occurs at all levels in the recursive object hierarchy in a uniform and consistent fashion.

2.5.2 Constraint Inheritance

An object's prefixes provide the initial set of constraints; these constraints are conjoined with constraints in the object's body. A derived object must satisfy all of the constraints of its prefixes as well as the ones it defines itself. New constraints that restrict the derived object's potential states are balanced by new attributes that expand its potential states.

We have already seen the use of prefixing as a convenient way to further specify a given object. In instantiating aPoint, we used aPoint as a prefix to a set of additional expressions defining the point's attribute values:

```
p: aPoint { x = 15; y = 20; };
```

Prefixing can also be used to extend an object with both constraints and attributes. For example, given the temperature object aCFTemp:

```
aCFTemp: {
   C, F: aNumber;
   F-32 = 9/5*C;
};
```

a new object can be easily defined to include the temperature in Kelvin, by specifying a new attribute as well as an additional constraint:

```
aKCFTemp: aCFTemp {
   K: aNumber;
   K - 273 = C;
};
```

This object is equivalent to the following:

```
aKCFTemp: {
   C, F: aNumber;
   F-32 = 9/5*C;
   K: aNumber;
   K - 273 = C;
};
```

All expressions in the prefix `aCFTemp` are simply included into the body of the base object. Because `aCFTemp` defines `C` as an attribute inside of the object, `aKCFTemp` may directly access it in its body to relate the temperature in Kelvin with Celsius.

2.5.3 Method Inheritance

Since methods may define side effects and contain other imperative statements, the model provides more control in their merging than simple conjunction or concatenation. We adopt Simula and BETA's `inner` operator to specify a point for the merging of a method definition with its corresponding prefix definition. We write `inner` in an object definition to indicate the point at which a derived object's code will be merged. This top-down merging is used to guarantee and control the inherited behavior of prefixes while allowing specialization to occur.

Sequencing in methods is indicated by the *bang* operator (`!`). An action `R` followed by an action `S` is denoted by the expression `R ! S`.

Using both the sequencing operator and `inner` we specify the order of actions in combined methods. For example, the method definitions

```
V: aMethod { R ! inner ! S };
W: V { X ! Y ! Z };              -- Derived from the method V
```

would result in W's method definition being equivalent to

```
W: aMethod { R ! X ! Y ! Z ! S };
```

since the body of W replaces the inner in V to form the combined body.

Thus, the combined body of a method is the prefix's body with the inner operator replaced by the method's body. In the absence of inner, the merging point is simply the end of the prefix body; if inner is present but there is no body to insert, inner becomes a no-op. In the presence of multiple prefixes, inner generalizes to wrap subsequent prefix expressions around preceding ones.

More specifically, prefixes merge together with the base to form a single, flat expression that is the conjunction of all the expressions in the prefixes and base. If an object A is prefixed by A1, A2, ... An with a body An+1,

```
A: A1, A2, ... An { An+1 }
```

then the system puts together recursively the combined definition A: A1, A2, ... An, An+1 using a PrefixMerge algorithm:

PrefixMerge of bodies A1, A2, ... An An+1 is defined as the concatenation of

- the expressions in the body of A1 before the inner operator;
- the PrefixMerge of bodies A2, ... An An+1; and
- the expressions in the body of A1 after the inner operator.

For example, the following object definition R is prefixed with X, Y, and Z:

```
X: aMethod { a ! inner ! b };
Y: aMethod { c ! inner ! d };
Z: aMethod { e ! inner ! f };

R: aMethod, X, Y, Z { g };
```

Since an inner in an expression inserts expressions in the following prefix's body, R's equivalent expression body would be { a ! c ! e ! g ! f ! d ! b }:

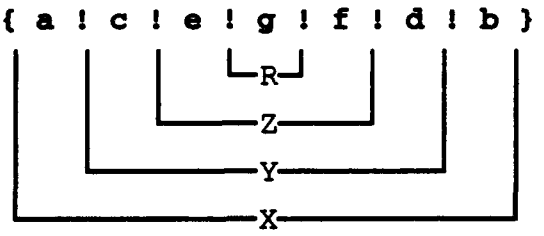


Figure 2-9. R **derived from** X, Y, **and** Z **using** inner.

To illustrate a use of inner, consider an object called aWindow that defines a draw method. This method sets the clipping region before drawing, draws its contents, and then restores the clipping region. The designer of aWindow thoughtfully included an inner so that objects using aWindow as a prefix could take advantage of the fact that the clipping region had been set. Placing inner in a constraint pattern is a top-down design decision that the programmer is responsible for; he must choose the location for inner such that potential refinements of the constraint pattern may take advantage of the structure and partial state of the pattern's methods when the refinement is called. To do this, we know that any refinements of aWindow must draw their contents only after the clipping region is set, to restrict drawing to the window area. We then define draw in aWindow as follows:

```
aWindow: {
   ...
   draw: aMethod { setClip ! drawContents ! inner ! restoreClip };
   ...
};
```

Now, consider two specializations of aWindow, aTitledWindow and aBorderedWindow, both inheriting from aWindow:

```
aTitledWindow: aWindow {
   ...
   draw: aMethod { drawTitle };
   ...
};

aBorderedWindow: aWindow {
   ...
   draw: aMethod { drawBorder };
   ...
};
```

Inheriting from both `aTitledWindow` and `aBorderedWindow` results in the following:

```
myWindow: aTitledWindow, aBorderedWindow {
   draw: aMethod { drawMyContents };
};
```

where the sequence of operations when `draw` is invoked on `myWindow` would be

```
{ setClip ! drawContents ! drawTitle ! drawBorder !
   drawMyContents ! restoreClip };
```

Note that if we had defined `myWindow` with the two prefixes in the opposite order, as

```
myWindow: aBorderedWindow, aTitledWindow {
   draw: aMethod { drawMyContents };
};
```

the sequence of operations when `draw` is invoked would be

```
{ setClip ! drawContents ! drawBorder ! drawTitle !
   drawMyContents ! restoreClip };
```

resulting in the border being drawn before the title.

2.6 A Simple Example

To illustrate use of the model more fully, we define a set of objects for controlling the temperature in a house. These objects include our previously-used object `aTempSensor`, whose value is the current temperature in degrees Kelvin; `aFurnace` that can be turned on and off; and `aThermostat` that communicates with these objects using constraints and messages. The previously-defined object `aKCFTemp` is used as an attribute inside of `aThermostat`.

As we saw in section 2.3.2, `aTempSensor` has a single attribute, its `value`. Since the sensor is a primitive object, it returns only a number that is interpreted by the programmer as a temperature in degrees Kelvin.

The furnace has as state a single Boolean value `on` that is `true` if the furnace is on and `false` if it is off, and two methods that change its state:

```
aFurnace: {
   on: aBoolean;      -- Is the furnace on?

   turnOn: aMethod    { --turn on the furnace-- };
   turnOff: aMethod   { --turn off the furnace-- };
};
```

Figure 2-10. aFurnace.

Finally, the thermostat object has references to a temperature sensor and to a furnace, so that it can turn the furnace off and on in response to temperature changes within the house.

```
aThermostat: {
   -- Visible owned attributes
   currentTemp, desiredTemp: aKCFTemp;

   -- Hidden shared attributes
   #furnace:    ^aFurnace;
   #sensor:     ^aTempSensor;

   -- Thermostat message: setting the desired temperature in degrees C.
   "setDesiredTemp t'aNumber": aMethod { desiredTemp.C = t; };

   -- Change interface: what to do when objects change.
   "sensor.value changed": aMethod { currentTemp.K = sensor.value; };

   "desiredTemp changed", "currentTemp changed": aMethod {
      if furnace.on & (currentTemp.C >= desiredTemp.C) thenDo furnace.turnOff;
      if furnace.off & (currentTemp.C < desiredTemp.C) thenDo furnace.turnOn;
   };
};
```

Figure 2-11. aThermostat **with** changed **messages.**

In this example, there is a single method defined for external clients of the thermostat. This method, "SetDesiredTemp t'aNumber", **allows clients to set the desired temperature of the house. Clients may use the visible attributes of** desiredTemp **and** currentTemp **to read the desired and current temperatures respectively in degrees K, C, or F.** changed **messages notify the object of changes that occur to** sensor, desiredTemp **and** currentTemp. **In the case of the sensor, the current temperature, an instance of** aKCFTemp, **is updated by setting its Kelvin attribute to the same value as the sensor's value. This causes the** C **and** F **attributes of** aKCFTemp **to be recomputed when their values are needed. In the case of either** desiredTemp **or**

currentTemp changing, the furnace's on attribute is checked, and depending on the temperatures, a message is sent to the furnace to either turn it on or off.

Note that constraints on owned attributes of the thermostat maintain the state's consistency. In contrast, the message interface between independent objects is used to allow object encapsulation and to take advantage of strict interface definitions.

If, instead, the furnace and sensor were an integral part of the thermostat system and not shared with any other thermostats, we could simplify the thermostat code by using constraints only, with no messages:

```
aThermostat: {
   -- Visible owned attributes
   currentTemp, desiredTemp: aKCFTemp;

   -- Hidden owned attributes
   #furnace:    aFurnace;
   #sensor:     aTempSensor;

   -- Thermostat message: setting the desired temperature in degrees C.
   "setDesiredTemp t'aNumber": aMethod { desiredTemp.C = t; };

   currentTemp.K = sensor.value;
   furnace.on = (currentTemp.C < desiredTemp.C);
};
```

Figure 2-12. aThermostat with internal constraints.

In this case, constraints are used to update continuously the current temperature given the sensor value, and the state of the furnace based on the current and desired temperatures.

2.7 Implications of the Model

This model departs from traditional object-oriented programming and constraint programming languages in a variety of ways.

2.7.1 Separation of Side-Effecting and Non-Side-Effecting Interfaces

While standard object-oriented programming provides operations for both accessing and changing objects, this model separates operations that do not modify state (i.e., attributes) from ones that do (i.e., messages). The model relies on attributes as being measurable traits of an object. Having such attributes explicitly available makes it possible to specify the effect of a given side-effecting operation in terms of the object's attributes: which attributes will receive

new values, which may change, and which must stay the same.

Given the ability to directly name measurable traits of an object via attributes, the model can take advantage of the consistency constraints to help generate satisfaction methods. To generate a satisfaction method in the case of a simple attribute, the value can either be returned directly as a stored state value, or it can be computed as a function of the other object attributes. However, in the case of a state-changing method, we need more information about how the object is to be updated. Since the attributes refer to measurable traits of the object, we can specify that certain attributes are not to change during the resatisfaction process; all other attributes are then free to change, and the constraints that relate the attributes to each other can be solved in a way specified by the method. If the system of constraints is underconstrained, then the value returned is a symbolic result.

2.7.2 Constraint-Based Method Definition

Rather than providing only imperative code in methods to modify an object, the model supports a mixed imperative and constraint-based approach. Defining methods in this way provides several advantages; most importantly, it can in some cases more closely approximate the cognitive model used by the programmer.

In standard object-oriented programming, methods may either return a function of attributes, which is a read-only operation, or they may cause the state of the object to be perturbed in a way that respects the consistency specification of the object. The first case is simple. The object's internal state is either already consistent, and the function is simply computed, or it is partially consistent, and other attributes must be updated before the final result is returned to the client. In the second case, however, an arbitrary number of attributes must be updated and their relationships encoded in the method. This can be prone to error; when the implementation changes, all methods that relied on the specifics of that implementation must be identified and recoded manually by the programmer. Thus, in standard object-oriented programming each method is separate, and has a responsibility to compute whatever attributes are necessary to maintain the object's implicit consistency requirements. The consistency specification for an object is distributed among all of the methods of the object, and there is no one location for the specification to be located.

In our model, we encode the object's methods so that the relations between the attributes are maintained automatically. The relations need not be updated continuously—for example, a particularly difficult attribute to compute can be computed lazily, when a particular message is received—but from the client's point of view, all of the object's attributes are consistent with respect to the object's model. For example, in the `aKCFTemp` object, if the value of `K` changes, it is not necessary to recompute `C` and `F` immediately; this can be done lazily when `C` and `F` are requested by a client.

Finally, our mechanism allows for direct declaration of constraints between object attributes in a single location in the object definition, rather than distributed throughout many different methods. In addition, our model provides specific encoding of the intent of the method by describing how the measurable characteristics of the object, its attributes, should change.

2.7.3 Restricted Pointer Manipulation

Most object-oriented languages allow unrestricted manipulation of objects via references, or pointers. The constrained objects model restricts the role of pointers so that they are used only for objects that are shared with others.

References to shared objects are not a problem in standard object-oriented programming. When a method is executed (e.g., a message is received), the object computes its new state by inspecting the state of all of the shared objects and using that state in its recomputation process. Shared objects are useful for both communication between objects and for saving space in the system.

Having references in a constraint-based object-oriented system, however, brings up a severe problem. A pointer from an object `A` out to another object `B` that is shared creates an additional mechanism for changing `A`'s state. Normally in object-oriented programming, changing `B`'s state does not change `A` directly, since `A` simply has a pointer to `B`; in contrast, the requirement that `A` be maintained consistent with respect to `B` in a constraint language forces `A` to be resatisfied whenever `B` changes. Thus, pointers from an object to another create an additional interface to that object, in effect, making `B` a part of `A`'s interface. In our model, we do not allow such interfaces, since they interfere with encapsulation.

Another issue involving pointer manipulation has to do with assignment of a shared object to a variable. If we allow constraints to specify relationships between an external object and internal attributes, changing the object to which particular variable refers can cause an arbitrary amount of resatisfaction to be required. Not only will the object have to be resatisfied due to different state in the external object, but if the new external object has a different implementation than the old one, it is potentially necessary for the entire object to be rereduced to incorporate the changes. For example, recall that there is a shared attribute `desktop` that points to an instance of `aWindowMgr` in `aRectangle`'s definition. Assume that we have also declared a constraint between the `desktop`'s center and the rectangle's center, so that the rectangle is always centered in the `desktop`:

```
aRectangle: {
   ...
   desktop: ^aWindowMgr;
   ...
   center = desktop.center;
};
```

When the value of `desktop` is first set to an instance of `aWindowMgr`, the rectangle can be fully satisfied. However, if `desktop` is changed subsequently to point to another object that is a descendent of `aWindowMgr`, the rectangle must be resatisfied, since the new `desktop` referent could have completely different state, and implementation, from the old `desktop` referent. The convenience of pointer manipulation thus presents a difficult problem: how to minimize recomputation when pointers are changed? We also need to minimize recomputation in cases where an object changes structurally, such as when a new subpart is added or a constraint is changed. While the model itself does not address minimizing recomputation in these circumstances, we discuss in Chapter 5 several strategies that can be implemented to handle this situation, if the model is extended to allow this behavior.

In our model, references to shared objects are allowed in a limited fashion. Constraints are not allowed on a shared attribute. This restriction avoids the problem of having to resatisfy the entire object when a shared object's value changes, and also avoids the problem of unsafe modification of the internals of an object. However, methods defined in an object may access attributes of a shared object during method invocation. When the shared object changes, the system can notify all of its dependents, allowing them to update their own state by accessing the shared object's attributes at that time.

2.7.4 Strict Top-Down Inheritance

Traditional object-oriented programming provides bottom-up inheritance with overriding of methods. Our model provides a strict top-down inheritance mechanism with, initially, no method overriding. This mechanism, which follows the Scandinavian tradition of object-oriented programming exemplified by Simula [Dahl70] and BETA [Kristensen89], creates derived objects that are pure extensions of their prefixes. An object does not override the attributes of its prefixes; instead, its attributes incorporate all of the definitions of their prefix namesakes. The result of this mechanism is that inheritance of objects is strict; in the absence of overriding, constraints and attributes that are defined in ancestors are always guaranteed to exist in the derived object. While the set of valid states for such a derived object will most likely be different, they will always be a strict subset of the set of valid states for any prefix. More specifically, the set of valid states for a derived object will be a non-strict subset of the intersection of the valid states of each of its prefixes. Note that redundant inherited expressions are of no consequence, since any equivalent expressions in the combined body are automatically discarded, while mutually inconsistent expressions can often be detected during the constraint satisfaction process.

Strict inheritance is safer than inheritance with overriding in that derived objects preserve properties that they inherit. This is quite different from typical object-oriented languages that allow overriding in a bottom-up fashion, where objects do not necessarily preserve inherited properties.

Also, unlike typical object-oriented programs, objects in this model inherit only constraints, attributes, and method definitions; the actual implementation of each derived object is

customized, based on the newly-combined specification. In standard object-oriented programming, methods call other methods defined and compiled in superclasses to perform merged behavior. In this model, the specifications of all namesake methods in prefixes and the base are merged and reduced to a single customized method that can take advantage of local information in the derived object.

2.7.5 Restricted Constraint Solving to a Single Object

This model does not describe a general-purpose constraint satisfaction mechanism, nor is it intended to do so. Other constraint models allow constraints at any level within the system. We provide constraints only within an encapsulated object, while messages are used exclusively between such objects. By choosing to restrict the constraint solving to the scope of a single encapsulated object, the model addresses particular issues quite differently from other constraint systems. By focusing on a simple dichotomy of separated messages and constraints, we gain the following advantages:

- We retain an important benefit of object-oriented programming: since the object-oriented message-passing style limits interaction of objects to a small interface, very large systems can be built with this model.

- Each object, being a constraint system in its own right, partitions the program into small, easily solvable sets of equations. This helps reduce the complexity of the constraint satisfaction problem by explicitly partitioning the constraint space.

- Satisfaction methods can be compiled for each message, using local reasoning, for solving the object's internal constraints in a controlled way.

- Debugging is potentially simpler, since constraint dependencies are local to encapsulated objects, simplifying the tracing of effects.

- Since an object defines a constraint system in its own right, each object may include a specialized constraint solver for the particular kinds of constraints being solved in its encapsulated scope.

However, there is one significant disadvantage as well:

- The two-level organization, constraints below and messages above a certain abstraction line, is not as general as a complete mixed messages-and-constraints system such as Kaleidoscope [Freeman-Benson90]; this forfeits the ability to have arbitrary constraints among objects at any level.

Systems such as Kaleidoscope and ThingLab allow arbitrary changes to attributes of objects, as well as constraints at all levels within the system. The way that the satisfaction process is directed is through a hierarchy of constraints that come into play as particular attributes are

given values. Rather than having a strict interface with an explicit specification of how a particular side-effecting message should be handled, different strengths of constraints on parts of the object are used instead to direct the satisfaction process more generally. For example, instead of explicitly fixing the `width` and `height` in `aRectangle` when moving its `center` as would be done in our programming model, a Kaleidoscope program might put object-level constraints on `width` and `height` that keep them from changing unless other, stronger constraints becomes applicable, such as during assignment. A very flexible specification of constraints is possible, with default behaviors and prioritizing of constraints easily handled.

While being more powerful, the Kaleidoscope approach has several disadvantages. The first is that constraints are not bound closely to objects, and it is difficult to see textually what constraints are in force for a given object. Secondly, Kaleidoscope suffers from the constraints-everywhere approach in that the interface to an object is not clearly defined, since any constraint on the object may cause its state to change. Finally, the very flexibility that constraint hierarchies gives the programmer can make it difficult to code explicitly how one would like a particular perturbation to change an object's state.

In addition, our model, by separating the constraint systems into encapsulated objects, can potentially be enhanced to provide concurrent evaluation of object reduction. Since the interfaces between encapsulated objects are strictly through message-passing, we can use techniques developed in the object-oriented programming field to allow individual objects to reduce concurrently.

2.8 Model Summary

This model provides object-oriented programming with constraints via a two-level organization. Encapsulated objects consist of subobjects that communicate among one another solely by constraint expressions, while encapsulated objects communicate with other encapsulated objects by message passing.

An object is defined as:

- **Attributes:** parts of the object that encode state and provide an interface for measurable traits;
- **Constraints:** expressions that define valid object states; and
- **Methods:** operations that define specific ways to perturb the state while respecting the object's internal constraints.

Objects send messages to other objects to request values of attributes or to invoke methods. Methods are defined as a set of temporary constraints that must hold during method evaluation, along with operators to fix attributes to their existing values and to assign new values to attributes. Encapsulation with a message interface allows the object to maintain complete

control over its own internal state.

A shared attribute provides a mechanism for sharing a single object among a group of dependents. The system sends messages to notify the dependents when the shared object changes. It is the responsibility of each dependent to perform whatever actions are necessary to incorporate any state changes of the shared object into its own state.

Finally, an object may inherit attributes, constraints, and methods from any number of ancestor objects, called prefixes. The inheritance process merges together attributes; attributes with namesakes in other prefixes or the object itself are recursively merged together to create a single derived attribute. The derived object's constraints are simply the conjunction of its own constraints and those of its prefixes. Each method in the derived object is either inherited directly from prefixes, or merged together, top-down, with Simula's `inner` operator to control the order of merging.

This model provides a variety of advantages over object-oriented and constraint programming languages. While retaining the most important benefits of object-oriented programming, encapsulation and inheritance, the model also provides constraints for automatically holding the internal state of objects consistent. Methods in this model use the constraint mechanism to control object modification, and specify only those constraints that are required for the given method. Finally, since each object is a small constraint system in its own right, we can partition the program into small, easily-solvable sets of equations, thus reducing the complexity of the constraint satisfaction problem.

Chapter 3
Constraint Patterns in Siri

This chapter describes Siri, an object-oriented language that implements the constrained objects model of Chapter 2. A unique feature of Siri is its sole abstraction mechanism called a *constraint pattern*.

The purpose of designing Siri is three-fold:

- To help validate the constrained objects model;
- To explore the kinds of programs that can be written with the model;
- To experiment with generalizing object-oriented features through the constraint pattern abstraction;
- To investigate the possibilities of a single abstraction mechanism.

Most programming languages have many different kinds of code and data abstractions. Object-oriented languages in particular provide classes, metaclasses, methods, and control structures, each with their own naming and scoping rules; features such as multiple inheritance and constraints introduce even more complexity. Our goals for Siri are to support the constrained objects model fully, with a language that is as simple and flexible as possible. In Siri, constraint patterns play the roles of code and data abstractions, and subsume the traditional object-oriented constructs of classes, instance variables, methods and control structures.

The constraint pattern is a blending and extension of features found in two existing languages. The first, BETA [Kristensen89], is an object-oriented language in which one specifies programs using hierarchical block-structured object descriptors called *patterns* (thus the name "constraint pattern" in Siri). We can think of Siri's constraint patterns as constraint-oriented versions of BETA's pattern abstraction, with modified syntax. The second, Bertrand [Leler88], is a constraint language based on *augmented term rewriting (ATR)* in which one uses rules that implement an equation solving substrate for constraint satisfaction. Siri evaluates constraint patterns using a variant of augmented term rewriting called *ATR+*, and uses Bertrand's equation solving mechanism for a variety of purposes: constraint satisfaction, method generation, consistency checking of inheritance structures, and compilation.

In Siri:

- Nested constraint patterns define a fully hierarchical object structure; each object defines a naming environment for creation of subobjects.
- Objects are related in a prototype-child model [Chambers89]: any object may serve as a *prototype* of another for inheritance, and an object may inherit characteristics from any number of existing objects.
- The type and object concepts are unified; each object defines a type of the same name. However, Siri also provides the ability to define a separate type hierarchy that may be used by casting objects into compatible types.
- Objects communicate with each other using two distinct mechanisms: message passing and constraint satisfaction. Message passing objects consist of subobjects that communicate solely through constraint expressions.
- Nested subobjects serve multiple roles as instance variables, read-only attributes, and state-modifying method definitions.
- Special constraint patterns define control structures for looping and sequencing; programmers may define additional control structures.
- Programmers can create customized syntaxes by defining new operators with constraint patterns.
- Intrinsic constraint patterns define an equation solving mechanism for manipulation of algebraic expressions; programmers may define additional patterns for describing and manipulating expressions over other domains.

Each of these concepts will be explained in this chapter with examples.

The form of a constraint pattern is simple; hence, there are very few concepts to be discussed, though each has an important role to play in the abstraction.

In section 3.1 we will outline the basic concepts of term rewriting; augmented term rewriting; and our extension of augmented term rewriting, ATR+, that includes features for the implementation of an object-oriented language. ATR+ provides the execution model for Siri's constraint pattern abstraction. Section 3.2 describes briefly the syntax of the constraint pattern, and section 3.3 discusses the concept of an object as type, which is used for matching of typed variables to objects, a fundamental operation in constraint pattern evaluation. Following sections will describe each of the constraint pattern's parts in detail: section 3.4 outlines the *label*, used for naming objects and matching against expressions; section 3.5, the *prefix*, used for inheriting characteristics of existing objects; and section 3.6, the *body*, used for specializing

object definitions. Because each of these parts refers to and interacts with the others, there will be references to concepts that have not yet been discussed. Each reference will provide a short explanation of the concept while deferring the full description to the relevant section.

Once the constraint pattern abstraction has been introduced, the rest of the chapter will discuss the consequences of the constraint pattern design, and its origins. Section 3.7 will describe various uses of the constraint pattern; and finally, sections 3.8 and 3.9 discuss its limitations and advantages.

3.1 Term Rewriting Mechanisms and ATR+

Siri evaluates constraint patterns using a special term rewriting mechanism called ATR+. In order to provide a general understanding of ATR+ and some insight into how ATR+ is used by Siri, we first describe standard term rewriting and Bertrand's augmented term rewriting mechanism from which we derive ATR+.

Term rewriting (TR) is an evaluation mechanism that uses rules for the transformation of expressions in a particular domain. *Augmented term rewriting* (ATR) is standard term rewriting with the ability to bind values to variables. Siri's evaluation mechanism ATR+ is augmented term rewriting with prefixing, nesting, imperative behavior, and the concept of `self`, required features for object-oriented programming.

3.1.1 Standard Term Rewriting (TR)

Term rewriting is a process for transforming expressions by matching and replacement. The process transforms a *subject expression* by applying a set of *rewrite rules*. Each rewrite rule is a pair of expressions: a *left hand side*, and a *right hand side*. Rewrite rules are typically written

```
leftHandSide → rightHandSide
```

The term rewriting process consists of repeatedly searching for a *reducible expression (redex)* in the subject expression that matches the left hand side of a rule; when a redex is found, the matching rule is applied, replacing the redex with an equivalent, but transformed, expression. Both the left hand side and the right hand side may contain variables; when the left hand side matches a redex, the terms that correspond to the variables in the left hand side are copied, and the right hand side is instantiated with the matching terms in the variables' place. For example, the rule

```
a/(b/c) → a*c/b
```

transforms the subject expression

```
45/(z/16)
```

to

```
45*16/z
```

since the subject expression's terms match the form of the rule, where `a` is `45`, `b` is `z`, and `c` is `16`. When all such redexes are matched and the corresponding rules applied, the rewriting process stops.

Term rewriting systems must be designed so that they are *confluent* and have the appropriate termination properties. A term rewriting system is confluent if the order in which the term rewriting rules are applied to a subject expression does not matter; a given subject expression is always reduced to the same irreducible result, if such a result exists. To ensure confluence, several restrictions may be placed on the forms of rules (see [Hoffman85] for details on these restrictions):

- No variables may be introduced in the right hand side of a rule that do not appear in the left hand side.
- Two different left hand sides may not match the same redex if they give different results.
- A variable in a rule's left hand side may not appear more than once; this would require tests for equality that may be computationally intractable.
- Left hand sides in different rules may not overlap each other; the left hand sides `x(y(z))` and `y(z(t))` overlap in the redex `x(y(z(r)))`, and using one rule's transformation instead of the other may give a different result.

A term rewriting system must also terminate if there exists a sequence of rule applications that results in an irreducible expression. To help guarantee this behavior, Siri, Bertrand, and Hoffman and O'Donnell's equation solver place another restriction on the form of rules, *leftmost-outermost reduction* and *strict left sequentiality*, in addition to the four restrictions above. Leftmost-outermost reduction means that leftmost redexes are reduced before redexes on the right, and outer redexes before inner redexes. Strict left sequentiality means that we must always be able to determine whether we have found a match for a rule when traversing an expression tree depth-first, left-to-right in its terms before we go on to the next node of the tree. An additional consequence of this last restriction is that we can compile the left hand sides of rules into a table and perform a very fast left-to-right string matching process to find reducible expressions (see section 5.3.1).

An excellent survey of term rewriting issues can be found in [Dershowitz85].

3.1.2 Augmented Term Rewriting (ATR)

An augmented term rewriting system adds several features to term rewriting in order to perform constraint satisfaction. In this section we use Bertrand syntax and semantics to explain ATR, and in the following section we use Siri syntax and semantics to explain ATR+.

First, to be able to handle more than one expression at a time in a single subject expression, ATR allows a set of distinct expressions to be grouped together using operators to partition them. In particular, the semicolon operator separates expressions, and distinguishes between an assertion of an expression as being `true`, and a simple indication of its value. Thus, `x=14` is a statement which may be `true` or `false`, while `x=14;` asserts that `x` is in fact `14`. Because the semicolon is an operator itself, rules may match against it, which allows the rewriting process to merge, reorder, and solve equations. ATR then adds a single-assignment binding operation to allow the assignment of values to variables; binding makes it possible to eliminate variables from the subject expression and thus to perform operations like Gaussian elimination. The variables may be typed, and are organized in a hierarchical namespace, allowing objects in the system to have named objects as parts. Coupled with a set of equation solving rules, these features allow ATR systems to solve circularly referenced equations, including simultaneous equations, and to perform general linear constraint satisfaction.

In ATR, the left hand side of a rule is called the rule's `head`, and the right hand side is called the rule's `body`. Following Bertrand syntax, a term rewriting rule

```
head → body
```

is written as

```
head { body }
```

where an ATR `head` is equivalent to a TR `leftHandSide`, and an ATR `body` is equivalent to a TR `rightHandSide`.

In ATR, objects and variables may be annotated by a type. In Bertrand we assert the type of variables in the head by annotating them with a single quote mark followed by the type name. In order to restrict expression matching to correctly-formed subexpressions, variables in the head of a rule match only instantiated rules (objects) of the appropriate type. Further, we may define types and organize them in a subtype hierarchy. The subtype hierarchy provides a specificity ordering for matching rules against redexes, so that more specific rules match before less specific ones.

We may also type rules in the same way; this asserts that the result of applying the rule will be an expression, or object, of the given type. We type a rule by appending the type to the end of the rule, as follows:

```
head { body } 'type
```

We may use ATR rules to create new objects as well; the following creates an object of type 'point:

```
aPoint { x: aNumber; y: aNumber; } 'point
```

By using types to specify the kinds of objects that may be matched in the rule head, and to specify the type of the rule's result, we may write an ATR rule in Bertrand to add two point objects as follows:

```
pt1'point + pt2'point {
   newPt: aPoint;
   newPt.x = pt1.x + pt2.x;
   newPt.y = pt1.y + pt2.y;
   newPt
} 'point
```

This rule matches reducible expressions that consist of two point objects related by an addition operator. When the rule matches a redex of the appropriate form, it replaces it with the expressions in the rule's body, substituting the variables pt1 and pt2 in the body with the terms that they matched.

The rule body first creates a new point object by labeling aPoint with the new object's name, newPt. Labeling an existing object creates a copy of that object, creates a new name in the namespace hierarchy, and assigns the object to the label in the current namespace. Then, expressions equate the x and y coordinates of the newly-created point to the x and y coordinates of the point variables; using the semicolon operator asserts these equalities to be true. Finally, since the value of a semicolon expression is the value of its right argument, the value of the entire expression in the rule is newPt.

We can apply the point addition rule to the following expression, where p1, p2, and p3 are all point objects:

```
p1 + p2 = p3;
```

Matching and instantiating the rule results in the expression

```
newPt: aPoint; newPt.x = p1.x + p2.x; newPt.y = p1.y + p2.y; newPt = p3;
```

Note that the reason that we can consider an unasserted final expression to be returned is that expressions surrounding the original redex can use it as a term. As the other asserted expressions are rewritten away, the expression becomes the final assertion

```
newPt = p3;
```

which is the desired result.

3.1.3 ATR Extended for Constraint Patterns: ATR+

The extension of augmented term rewriting that Siri uses for constraint pattern evaluation is called ATR+. ATR+ provides a superset of ATR's functionality, but adds features necessary for object-oriented programming: inheritance, nesting, imperative behavior, and the concept of `self`.

In ATR+, constraint patterns are comparable to augmented term rewriting rules. Constraint patterns have a label (similar to a rule's left hand side, or an ATR's head), a body (similar to a rule's right hand side, or an ATR's body), and an optional type, like an ATR rule. However, constraint patterns extend ATR rules in five ways:

- **Prefixing.** Constraint patterns may be extended using BETA's prefix concept. Objects defined in prefixed patterns, along with constraint expressions, are imported and integrated, through the rewriting process, into the prefixed pattern.

- **Nesting.** While ATR uses a set of rules at a single level that are applied to a single subject expression, constraint patterns may be defined at multiple levels by nesting within each other. ATR+ evaluates expressions in a constraint pattern body using pattern definitions visible from the body. These definitions include definitions in the body itself, in its prefixes, and in surrounding scopes.

- **Delayed Evaluation.** ATR+ may delay the evaluation of a constraint pattern expression by surrounding it with square brackets, producing a delayed *block*. Using this delay mechanism, constraint patterns can control evaluation of such expressions by matching, adding and removing brackets as required. This procedural extension enables the control of evaluation order, and thus suitably-defined constraint patterns can provide imperative behavior such as sequencing and control structures.

- **Objects With State.** ATR+ unifies the distinct ATR concepts of rules and labeled objects into a single abstraction: the constraint pattern. Constraint patterns are not simply rules that transform expressions, but instead define objects with mutable state.

- **Partially-Reduced Objects.** While ATR does not evaluate expressions in a rule body until it is instantiated and becomes part of the subject expression, ATR+ may partially reduce constraint pattern bodies at any time, creating an object that may include references to `self`. Also, since constraint patterns refer to each other, and to themselves recursively, partially-reduced constraint patterns must be available at any point in the process for evaluating other constraint patterns. This is acceptable since a constraint pattern is valid at any point during the reduction; each step in the reduction process

preserves a similarity relation (equality).

These five extensions to ATR rules define constraint patterns in ATR+, which provide a variety of object-oriented programming facilities in a single abstraction.

We define the operational semantics of constraint patterns in terms of our extended evaluation mechanism ATR+, and thus a constraint pattern can be considered similar to an ATR rule with persistent state and identity. Since ATR+ is an extension of ATR, Siri's constraint patterns appear very similar to Bertrand rewrite rules. However, constraint patterns define objects with state that can be modified at runtime. In Siri, each constraint pattern body is considered a subject expression and can be reduced at any time. In Bertrand there is a single subject expression that is reduced using the rewrite rules. Reducing constraint pattern bodies is similar to precompiling bodies in Bertrand, with the exception that the resultant entity is a true object which can be referenced, created, and side-effected.

Constraint patterns also have several syntactic differences from Bertrand's ATR rules:

- If a constraint pattern's label is an expression rather than a symbol, quotation marks must surround the label;
- All labels, symbols and expressions alike, are followed by a colon;
- Objects serving as prefixes may precede the constraint pattern body;
- A semicolon terminates each constraint pattern; and
- Nesting of constraint patterns is permitted.

In Siri, we can code the constraint patterns to create a point object and to add two points as follows, with syntactic differences (in contrast to the Bertrand example in section 3.1.2) highlighted using bold text:

```
aPoint: {
   x: aNumber;
   y: aNumber;
   self                                  -- Returns the instance of aPoint
};

"pt1'aPoint + pt2'aPoint": aPoint {      -- aPoint defines type of self
   x = pt1.x + pt2.x;                    -- x and y refer to subobjects
   y = pt1.y + pt2.y;                    -- that are inherited from aPoint
   self                                  -- self returns instance of aPoint
};
```

The first constraint pattern defines an object called `aPoint`, with two coordinates, `x` and `y`. The second constraint pattern matches subexpressions that add two instances of `aPoint`, and returns a new instance of `aPoint` with its `x` and `y` coordinates being the sum of the corresponding coordinates of the arguments. Since this constraint pattern inherits from `aPoint`, we can express constraints about the instance's `x` and `y` attributes directly in the constraint pattern body.

Note that we do not need to specify a type, as we must in Bertrand, since the prefix `aPoint` defines the object type for us. Since any object can be used to create others just like it, it is natural to consider an object as a type definition as well.

3.2 The Basic Abstraction

Constraint patterns specify all objects in a Siri program. The most general form of a constraint pattern is as follows:

```
Label: Prefix { Body } 'Type;
```

This expression creates an object called `Label` that inherits from objects named in the `Prefix`, adds its own refinements in `Body`, and is of type `'Type`. A constraint pattern's colon, braces, and type quote can be read in English as "`Label` **is a** `Prefix` **including** `Body` **conforming to** `Type`".

The label can be thought of as the name of an *object*, and the part of the constraint pattern after the label, namely the `Prefix`, `Body`, and `Type`, is its *object descriptor*. The object descriptor is a specification of an object to instantiate. Depending on the pattern's form (the kind of label and the specific form of the object descriptor), objects play the roles of classes, instances, method definitions, and control structures.

In the object descriptor, only one of either `Prefix` or `Body` is required, and the `Type` is completely optional, since its role is played typically by the object's prefixes. So the following forms are also acceptable:

```
Label: { Body };
Label: Prefix;
Label: { Body } 'Type;
Label: Prefix 'Type;
```

Objects listed in the `Prefix` specify the initial characteristics of the object that are modified by expressions in the `Body`. Objects in the `Prefix` may in turn be defined using prefixes; thus, an object's *prefix set* is the set of all objects that are inherited, directly or indirectly, by that object. For constraint patterns that omit the prefix, Siri assumes the prefix `anObject`.

3.3 Objects as Types

In Siri, objects also define types; an object `R` that is a descendant of another object `S` via inheritance is considered to be of type `'S`. Thus, each object `S` in the system defines a corresponding type `'S`. A type specifies an object's external interface available to clients, and also indicates the roles that the object can play in different contexts. An object's type is permanent, and so cannot change over time.

While an object defines its own type, it also *conforms* to other types defined by the objects it inherits, known as its *prefixes*(see section 3.4). Since an object may inherit from several others, it may actually conform to several different types; such a type system is called a *predicate type system* [Pratt82].

3.3.1 Type Conformance

An object conforms to a set of types that is defined by itself and its prefixes. Specifically, an object *conforms* to a particular type `'T` if the object `T` appears in its prefix set, or if the object is actually `T` itself. A given object conforms to any of the types defined by objects in this set, because it inherits characteristics from all of them.

In the following, we use `Prefixes(S)` as the list of the direct prefix objects in `S`'s constraint pattern.

We define `TypeSet(S)`, the set of types to which an object `S` conforms, as follows:

```
TypeSet(S):     if S = nil then ∅ else
                { 'S } ∪ LTypeSet(Prefixes(S)).
```

where `LTypeSet(P)` is the union of the set of types of all objects in the list `P`:

```
LTypeSet(P):    TypeSet(head(P)) ∪ LTypeSet(tail(P)).
```

We define `ObjectsConformingTo('T)`, the set of objects that conform to a given type `'T`, as follows:

```
ObjectsConformingTo('T):
                { T } ∪ { S | T ∈ PrefixSet(S) }.

PrefixSet(S):   if S = Nil then ∅ else
                Prefixes(S) ∪ LPrefixSet(Prefixes(S)).

LPrefixSet(P): PrefixSet(head(P)) ∪ LPrefixSet(tail(P)).
```

where `PrefixSet(S)` defines the set of prefixes reachable starting at `S`; and `LPrefixSet(P)` is

the union of reachable prefixes from all objects in P.

Informally, the set of objects conforming to type 'T includes all objects that, directly or indirectly, include T as a prefix. Thus, an object S conforms to a type 'T, or Conforms(S, 'T), if

```
Conforms(S, 'T):  true if S ∈ ObjectsConformingTo('T)
                  false otherwise.
```

For example, in the geometric object definitions

```
aQuadrilateral:   anObject                          { ... };
aParallelogram:   aQuadrilateral, SidesParallel     { ... };
aRhombus:         aParallelogram, SideLengthsEqual  { ... };
aRectangle:       aParallelogram, SidesRightAngles  { ... };
aSquare:          aRectangle, SideLengthsEqual      { ... };
```

then an object defined by `sq: aSquare;` conforms to all the following types:

```
'sq, 'aSquare, 'aRectangle, 'aParallelogram, 'aQuadrilateral,
'SideLengthsEqual, 'SidesRightAngles, 'SidesParallel, 'anObject
```

since they are the set of objects in sq's prefix set, including sq. Although the object sq does conform both to aParallelogram and SideLengthsEqual, it does not conform to aRhombus, since aRhombus does not appear in its prefix set.

An object defined by `p1: aParallelogram` conforms to

```
'p1, 'aParallelogram, 'aQuadrilateral, 'SidesParallel, 'anObject
```

since they are the set of objects in p1's prefix set, including p1.

The *type signature* of an object consists of the names and types of the object's attributes. For example, the type signature of aPoint with numeric attributes x and y is

```
'aPoint
   x'aNumber
   y'aNumber
   ...plus all attributes of anObject...
```

A programming environment for Siri can store type signatures in the type objects themselves, and use them at compile time to perform checking on the object types in patterns.

3.3.2 Type Specificity

Siri partially orders types by *specificity*. We order pairs of types by determining if one type is a subtype of the other; if so, it is more specific. A type 'P is a *subtype* of another type 'Q if the object P conforms to the type 'Q, and the object P is not Q.

In general,

```
SubType('P, 'Q) = true if Q ∈ PrefixSet(P)
                  false otherwise.
```

Type specificity provides a partial order on parameterized labels that have the same structure. When a group of labels all match an expression, the matcher chooses the label that is most specific. See section 3.4.2 for parameterized label examples using type specificity in matching.

3.3.3 Role Types

Since an object might have multiple prefixes and thus multiple supertypes, it provides a set of distinct attribute interfaces. However, a client may prefer to view an object as having only one interface and thus only one type. For example, consider an object that prefixes aStream and aWindow, and thus defines both stream and window attributes; its definition would look like

```
aStreamingWindow: aStream, aWindow {
   ...
};
```

If another object, for example a network port, used an instance of this kind of object but only cared about the stream interface, it could declare it as follows:

```
aNetworkPort: {
   sw: aStreamingWindow 'aStream;
   ...
};
```

By affixing a role type to sw's object descriptor, the interface to the object is restricted to that of aStream. Attributes that are not defined in aStream will not be visible to sw's clients.

Similarly, if we would like to create two different implementations of aPoint, aPolarPoint and aRectangularPoint say, we could set their role types to aPoint. Users of these implementations would the only be able to access the particular attributes of aPoint, regardless of whether they defined additional attributes.

3.4 Labels

Labeling is the mechanism for naming and instantiating objects. The constraint pattern's object descriptor (i.e., prefix, body, and type) specify an object; labeling creates a new object based on the object descriptor in the enclosing object's environment.

Labels come in three variants: *simple labels*, *parameterized labels*, and *label repetitions*.

3.4.1 Simple Labels

A simple label is a string identifier that names an object. Examples of constraint patterns with simple labels include:

```
aRectangle: { ... };          -- Creates a rectangle object
aPoint: { ... };              -- Creates a point object
aSquare: aRectangle { ... }; -- Creates a square, inheriting from aRectangle
x, y, z: aNumber;             -- Creates three numbers
wm: ^aWindowMgr;              -- Creates a shared attribute to aWindowMgr
```

In each of these cases, the system creates and names new objects based on an object descriptor and its label. A group of objects may be created at once from the same object descriptor by providing a list of labels, as in the constraint pattern `x, y, z: aNumber;` that creates three instances of `aNumber`.

3.4.2 Parameterized Labels

Parameterized labels are expressions, enclosed in quotes, with free variables to which the constraint pattern body may refer. A parameterized label is used to define objects whose value is dependent on arguments in its label. More specifically, a constraint pattern with a parameterized label defines an object that is underconstrained. Since labels are the names of objects, parameterized labels denote a family of objects defined by expressions of a particular form; this is similar to a function with parameters.

The variables in a parameterized label are distinguished by being suffixed with their types. A parameter variable may match only those objects that conform to variable's type (see section 3.3.1). For example, the constraint pattern

```
"a'aFloat + b'anInteger": aFloat { ... };
```

denotes the family of objects that are defined by constraint expressions adding an instance of `aFloat` object to an instance of `anInteger` object, in that order. Such expressions include

```
2.2 + 5
3.14159 + 0
```

but not

```
3 + 4                          -- anInteger plus anInteger
"a" + 19                       -- aString plus anInteger
0 + 3.14159                    -- anInteger plus aFloat
```

Labels use types to denote what kinds of objects the variables may match in a redex. The following label denotes the sum of `aFloat` object with `anInteger` object, which results in `aFloat` object:

```
"a'aFloat + b'anInteger": aFloat { ... };
```

The variables `a` and `b` in the above label are used in the body to compute the object's value.

Since each variable in a parameterized label may be independently typed, labels may match expressions of arbitrarily mixed type. We can consider a parameterized constraint pattern as a kind of method; rather than simply sending the message `+` to a receiver that is of type `'aFloat`, and thus forcing the receiver to perform further discrimination on the argument to `+`, we examine the entire context of the expression in Siri (these are known as *multi-methods* in systems such as CLOS [Bobrow88]). The need to examine the types of multiple arguments is not at all unusual in user interface programs: situations such as adding two numbers of different kinds, or drawing a shape on a graphics device, require discrimination on the types of more than one argument to determine the appropriate meaning.

A parameterized label can be an arbitrarily complex expression. For example, we can write

```
"a'aNumber * (x'aNumber + y'aNumber) = z'aFloat; x": aFloat {
   a ~= 0; z/a - y };
```

which solves for `x` given this particular equality expression. This pattern could match, for example, the bold expression in

```
r, s: aNumber; 16 * (r+s) = 3.14; r
```

since `16`, `r` and `s` are known to be instances of `aNumber`, and `3.14` is `aFloat`.

Because many different labels may match the same expression, we use type specificity to provide a partial order on parameterized labels that have the same structure. When more than one label may match a redex, the most specific label is used. For example,

```
"a'aFloat + b'anInteger": aFloat { ... };
```

is more specific than

```
"a'aNumber + b'anInteger": aNumber { ... }
```

because `'aFloat` is a subtype of `'aNumber`. The redex

```
3.14 + 7
```

matches the pattern with label `"a'aFloat + b'anInteger"`, although the redex conforms to both constraint pattern labels.

In addition, parameters that appear earlier in the label are more significant than parameters that appear later; thus

```
"a'anInteger + b'aNumber": aNumber { ... }
```

is more specific than

```
a'aNumber + b'anInteger": aNumber { ... }
```

since `anInteger` is a subtype of `aNumber`, the two types that are first in the label expressions.

Not all pairs of types have a specificity relationship. For example, say `'aWindow` and `'aPort` are both subtypes of `'anObject`, but neither is a subtype of the other; thus, they cannot be compared. So the constraint patterns

```
"a'aWindow someOp b'aString": { ... };
"a'aPort   someOp b'aString": { ... };
```

cannot be ordered via type specificity. In this case, the objects themselves provide a specificity ordering by using the order of their prefixes. An object that is both `aWindow` and `aPort` would inherit from both objects in a particular order:

```
q: aWindow, aPort: { ... };
```

and the system uses this ordering to specify that `q` is more specifically `aWindow` than `aPort`.

3.4.3 Label Repetitions

As we have seen, labeling creates one or more objects based on an object descriptor. Sometimes it is useful to create a number of objects from the same object descriptor and give them a series of computed names, rather than listing each name literally. A label repetition creates an object from a single object descriptor that consists of a collection of subobjects, like an array, with generated labels to reference each part of the object. Any object may have a variable number of indexed subobjects; label repetitions provide a mechanism for creating, and accessing, these subobjects.

We use the at-sign operator in labels to indicate a label repetition. The general form of a label repetition is

```
labelName@subLabelExpr: objectDescriptor;
```

The `labelName` is the name of the object; the subobject labels are generated by evaluating the `subLabelExpr`, which returns a list of subobject labels.

The names in a label repetition may be numeric or symbolic. To illustrate, if we would like to create an object with four points as subobjects, indexed 1 through 4, we can write

```
myPoints@(1 to 4): aPoint;
```

This creates an object, `myPoints`, with 4 instances of `aPoint` as subobjects. This label repetition is equivalent to the pattern

```
myPoints: { @1: aPoint; @2: aPoint; @3: aPoint; @4: aPoint; };
```

Note that numeric labels begin with the indexing operator `@` to distinguish them from the values of the numbers themselves.

String labels are used directly; for example, the following constraint pattern generates an object with 26 rectangles, each labeled with a character:

```
lotsOfRectangles@("a" to "z"): aRectangle;
```

which is equivalent to

```
lotsOfRectangles: { a: aRectangle; b: aRectangle; ... ; z: aRectangle; };
```

A label repetition creates subobjects in the order defined by the repetition interval. In the case of numeric repetitions, the objects are ordered numerically, and in an alphanumeric repetition, the objects are ordered lexically. However, this is simply an effect of the expressions that generate the label names; we could create an object with names that are not ordered lexically by providing the subobject names explicitly out of order:

```
randomOrder@("x", "r", "t"): aNumber;
```

which would result in the object

```
randomOrder: { x: aNumber; r: aNumber; t: aNumber; };
```

The only ordering that is significant is indicated by the labels, and so the order of the

subobjects themselves is unimportant.

By providing label repetitions, Siri does not need special structures like arrays to create objects with indexed entries. Constraint patterns provide the special @ syntax for label repetitions and indexing to simplify the creation and usage of these kinds of object collections.

3.5 Prefixes

So far we have seen prefixes used to specify an object's preliminary specification, or as a kind of declaration. We have also seen that a prefix can be considered as a kind of prototype from which the body in the object descriptor inherits attributes and constraints.

The prefix concept in Siri provides the complete inheritance features of the constrained objects model. Objects may be related to each other in a prototype-child hierarchy, where any object can act as another's prototype; this ability to use existing object for inheritance avoids the conceptual division of a class-based system. Inheritance of attributes and constraint expressions is performed top-down; combination of namesake attributes in prefixes and the body are automatically combined using prefixing of their attributes and constraint expressions recursively.

Derived constraint patterns are strict extensions of their prefixes. Nested patterns may not be overridden, but must incorporate all of the behavior of their namesakes in the prefix. This is a safety feature in that it guarantees that behavior defined in a prefix will be expressed.

However, prefixing can be used for much more than a simple inheritance structure. Prefixes may act as classes and modules (see section 3.7.1), as declarations of abstract type and as abstract interface specifications (see section 3.7.6), and as mixins for multiple inheritance (such as `'SideLengthsEqual` in section 3.3.1). Because any object descriptor may have a prefix, objects that play the roles of methods as well as classes and attributes may be extended using the prefix facility. The general applicability of prefixes broadens the concept of inheritance to all kinds of objects in a Siri system, as in BETA.

3.6 The Pattern Body

The pattern body is the part of the object descriptor that provides the detailed information about the object's implementation. While a prefix provides an initial definition for an object, the body extends that definition, and extends the lexical namespace and evaluation environment of the object being created.

The pattern body includes three kinds of expressions:

- Constraint expressions, specifying constraints that are to hold among objects within the evaluation environment;

- Imperative expressions, specifying imperative actions to perform on the objects within the evaluation environment; and

- Nested constraint patterns, specifying new objects that are to be created within the lexical namespace and evaluation environment.

These three kinds of expressions specify the state and behavior of an object created with a constraint pattern definition.

3.6.1 Basic Expressions

Expressions within a constraint pattern body are similar to arithmetic and message-passing expressions in a language such as Smalltalk. Siri expressions are structured by operators, which are symbols or groups of symbols with parameter placeholders. Operators specify constraint relationships between objects that are maintained by the system, and also denote message-passing imperative sequences that the system evaluates.

Examples of typical expressions include

```
3+4 = g;
a*x^2 + b*x + c = 0;
offset = thisPoint.x + myRect.center.x;
self.top = 20; self.left = 50; self

myRect.(moveTo newPt) ! display.(setPattern pat) ! display.draw;
while (x<10) do (display.flash ! x is previous x + 1);
```

As part of the basic system, Siri defines a comprehensive set of arithmetic, functional, logical, and string operators, and fundamental evaluation operators for constraint satisfaction; additional operators may be added by programmers.

The constraint pattern syntax is based on operators as well. Siri defines constraint patterns simply as expressions that are structured by the special intrinsic operators: semicolon, colon, braces, and brackets.

3.6.2 Operators for Structuring Constraint Patterns

The *semicolon* operator asserts and separates expressions. The semicolon asserts that its left argument is true, and returns the value of its right argument. For example, the expression

```
a = b + c; c
```

asserts that `a` is the sum of `b` and `c`, and `c` is the return value of the entire expression. Multiple

expressions may be linked together using semicolons as well:

```
x+y = 15; x-y = 10;
```

The *colon* operator (:) follows the constraint pattern label, and separates the label from the object descriptor (e.g., the prefixes, body, and type).

Braces { } indicate the scope of the constraint pattern body. It follows the label and optional prefixes, and encloses expressions that define the object.

Brackets [ ] surround a block of expressions to delay their evaluation, analogous to a *thunk* in a strict functional language or a BlockContext in Smalltalk. With this operator, we can define sequencing, looping constructs, and other procedures where expressions are evaluated in a particular order. The order of evaluation can thus be controlled by evaluating expressions after they are taken out of their blocks via constraint patterns that match against brackets. Because these control structures cause procedural behavior, they are restricted to use in only method code. Section 3.7.7 explains how Siri uses this feature to define control structures for sequencing and looping.

These are the most basic operators used in constraint patterns. Other operators, such as those for indexing, sequencing, and subobject access, will be described in the following sections.

3.6.3 User-Defined Operators

Programmers may define operators using constraint patterns; operators may be infix, prefix, postfix, or mixfix, with definable arity, precedence, and associativity, allowing the creation of a wide variety of syntaxes. Operators do not specify their parameter types, thus allowing a single operator to be *overloaded*.

An operator definition is an instantiation of `anOperator` with a label that defines the operator's form. The label consists of identifiers separated by underbars ("_") that indicate parameter positions. The constraint pattern's body includes expressions for binding the values of the precedence, associativity, and operator type[1]. For example, to define an infix plus (+) operator with precedence 100 (less than the multiplication operator's precedence of 150) we write[2]

```
"_+_":   anOperator {   %%precedence is 100; %%associativity is 8;
                        %%opType is aNumericOp; self };
```

The body of the constraint pattern binds the operator attributes `%%precedence`, `%%associativity`, and `%%opType`, and returns the operator itself as its value. The operator type is `aNumericOp` which is a subtype of `aNumber` (see Appendix B).

---

[1]Primitive attribute names, by convention, begin with %%.

[2]We are investigating a more natural specification of precedence and associativity.

Similarly, we write a prefix factorial (`fact`) operator with precedence `10` as follows:

```
"fact_": anOperator  { %%precedence is 10; %%opType is aNumericOp; self };
```

Since factorial is prefix, there is no need to bind the `%%associativity` attribute.

Finally, we can define mixfix operators for intervals and Boolean `if/then/else`:

```
"from_to_by_":    { %%precedence is 5; %%opType is anExprOp; self };
"if_then_else_":  { %%precedence is 5; %%opType is anExprOp; self };
```

A function or procedure-like name can be considered as a unary operator that takes a structured parameter expression as its argument. For example, the name `threeAverage` in the constraint pattern

```
"threeAverage(a'aNumber, b'aNumber, c'aNumber)": aNumber { a+b+c/3 };
```

is a unary operator taking a list of three parameters as its argument. For the sake of convenience, we can define constraint patterns with functional-style operators and have the system automatically create a suitable operator definition.

Note that a list of parameters has a fixed structure, and is thus different from a variable list of values. To accommodate an argument that is a variable list of values, we use a single list parameter. For example, the function

```
"average numbers'aList": aNumber { sum numbers/length numbers };
```

averages the values in a list of numbers of arbitrary length.

Since we can define new operators and thus new expression forms, we can specify and evaluate expressions over domains that are not necessarily numeric. For example, in user interfaces, expressions involving regions (arbitrary shapes) are used for window drawing, clipping, scrolling, and so on. A region algebra written using constraint patterns could be used to simplify region expressions, for example.

3.6.4 `self`

Used in an expression, the symbol `self` refers to the object that defines the current evaluation environment. In an attribute definition, `self` refers to the attribute, while in a method definition, `self` refers to the enclosing object. `self` is used often as the last, unasserted expression in the body to return the object itself as the expression value. In our `point` addition example,

```
"pt1'aPoint + pt2'aPoint": aPoint {
   x = pt1.x + pt2.x;
   y = pt1.y + pt2.y;
   self
};
```

self is the last expression in the object body, thus returning the created instance of aPoint with the appropriate constraints on the x and y attributes.

Because returning self is such a common occurrence, Siri allows some shorthand notation for returning self as the value of an object. For example, the following definitions are considered to be equivalent:

```
width: aNumber { right - left };                  -- Implicit use of self
width: aNumber { self = right - left; };          -- Assumed self returned
width: aNumber { self = right - left; self };     -- Fully explicit self usage
```

In the last, most specific case, self is representing the object width.

Similarly, numeric and string literals can be written directly, without their prefixes and without having to equate the object to self. The following definitions are also equivalent to each other:

Integers

```
n: 16;
n: anInteger { 16 };
n: anInteger { self = 16; };
n: anInteger { self = 16; self };
```

Strings

```
str: "Siri";
str: aString { "Siri" };
str: aString { self = "Siri"; };
str: aString { self = "Siri"; self };
```

3.6.5 Constraint Expressions

Constraint expressions state a relation between object attributes that must be maintained. Constraint expressions are often a natural way to specify a program. For example, the following word problem from [Leler88] is particularly amenable to specification using constraints:

"I have 25 coins in my pocket: nickels, dimes, and quarters. They are worth $3.45. I have 7 more nickels than dimes. How many coins of each type do I have?"

We can solve this directly, using a constraint pattern in Siri:

```
aWordProblem: {
   nickels, dimes, quarters: aNumber;

   -- Constraint expressions
   nickels + dimes + quarters = 25;
   nickels*5 + dimes*10 + quarters*25 = 345;
   dimes = 7 + nickels;
   nickels, dimes, quarters
};
```

Here, we have created three attributes of `aWordProblem` (`nickels`, `dimes`, `quarters`) using nested constraint patterns, and related them to each other via arithmetic constraints. The value of `aWordProblem` is the list `5, 12, 8`, corresponding to the number of nickels, dimes, and quarters in the solution.

Siri's constraint satisfaction mechanism can solve simple algebraic problems like these directly. Siri's solver is a linear and near-linear algebraic solver that can handle simultaneous equations using Gaussian elimination.

Near-linear expressions are nonlinear expressions that can still be handled by the solver. This is done by solving the linear equations first; by doing this, some of the nonlinear equations may eventually be solved by finding a value that transforms them into linear ones. For example, the three simultaneous equations (from [Leler88])

```
p * q = 10;
q + r = 3;
q - r = 1;
```

can be solved by handling the two linear equations first, and then replacing `q` with its value, thus making the first equation linear.

Although Siri's solver currently is limited to this class of equations, there are several ways to strengthen it; see section 5.5 for details on improving the solver.

3.6.6 Imperative Expressions

Method objects may include expressions for performing imperative actions. These actions may modify another object's state by sending it messages, if the object is encapsulated, or by assigning new values to the object's attributes, if the object is part of a constrained system.

Just as in languages like Smalltalk, methods can express sequences of purely imperative actions by sending messages. In Siri, imperative steps are separated by the sequencing operator `_!_`; we send messages to objects by using the dot operator `_._` to specify a receiver, or environment, in which a message expression is to be evaluated. For example, a purely

imperative method to back up a disk might be coded as follows:

```
backupDisk: aMethod {
   disk.makeConsistent !
   backupDisk.(copyAllFrom disk) !
   user.(notify "disk backup successful.");
};
```

This code first sends the `makeConsistent` message to the object `disk`; the method then sends the message `copyAllFrom disk` to the `backupDisk` object; and finally, the user object receives the message `notify "disk backup successful.".`

Within an encapsulated object, however, methods typically describe constraints for changing the attributes of the object. For example, imagine a constraint pattern that describes an airplane wing. This pattern might look like

```
aWing: {
   -- Wing attributes
   chord:          aNumber;
   span:           aNumber;
   area:           aNumber;
   taper:          aNumber;
   aspectRatio:    aNumber;

   ...constraints on attributes defining the wing's characteristics...
};
```

If we want to change the `span` of the wing but keep its `taper` and `aspectRatio`, we could define a method in `aWing` as follows:

```
aWing: {
   ...
   "changeSpanTo s'aNumber": aMethod {
      taper, aspectRatio fixed; span = s;
   };
   ...
};
```

This method changes the `chord` and `area` in response to the change in `span`, while keeping the `taper` and `aspectRatio` the same. The imperative action of assigning the `span` to the new value `s` combines with the constraint expressions fixing the `taper` and `aspect` ratio to perform an imperative result that is consistent with the object's internal constraints.

3.6.7 Expression Types

An expression's parsed representation is a tree with operators at the nodes, and subexpressions and individual objects as the nodes' children. The operator at the root of the expression tree is the *root operator*.

The type of an expression is the type of its root operator. Any given expression must eventually evaluate to an object of its expression type. By using the expression type as an indicator of the expression's final value type, we can match the expression as if it were an object of that type. For example, in the definition of the + operator, we defined the operator to be of type `aNumericOp`, a subtype of `aNumber`. The system can then consider expressions with + at the root of the expression tree to evaluate to `aNumber` of some kind. Thus, patterns like

```
"n'aNumber * 0": aNumber { 0 };
```

will match the entire expression

```
(4*5+6*7*8)*0
```

where the variable `n` matches the expression `(4*5+6*7*8)`. Since the precedence of + is less than the precedence of *, the parser groups the expression as `(4*5)+((6*7)*8)`, thus making + the root operator. The type of + is `aNumericOp`, and thus the entire expression can be matched as if it were `aNumber`.

Expression and operator types are heavily used by the system to manipulate expressions for constraint solving. Being able to type subexpressions by their operators provides a way to structure arbitrary expressions for matching. In addition, typing allows the system to match variable-length expressions, since the root operator can act as a proxy for the entire expression regardless of its length. Once structured using types, constraint patterns can easily transform and manipulate these expressions by matching with labels that have appropriately-typed parameter variables.

3.6.8 Nested Patterns

We have already seen that nested constraint patterns in an object's body recursively define subobjects. These subobjects define internal attributes that generally are accessible to clients, thereby defining an object's interface as well as its internal state description. These attributes internally perform the roles of instance variables, class variables and methods (see section 3.7).

For example, we can define `aPoint` as follows:

```
aPoint: {
   x: aNumber;
   y: aNumber;
   self
};
```

In this case, `x` and `y` are simple labels within `aPoint`, and are accessible to clients externally as attributes of `aPoint`, and internally within `aPoint` for constraints, if any.

Constraint patterns allow the definition of expressions relating attributes at any scope level in the hierarchical organization of an object. These expressions constrain the values of attributes at their level, which can then constrain values at lower levels. For example, consider the following object, `computeFEqualsC`, whose value is the temperature at which degrees Fahrenheit is the same as degrees Celsius:

```
computeFEqualsC: {
   aCFTemp: {
      F, C: aNumber;
      F = 9/5*C + 32;
   };

   A: aCFTemp;
   A.F = A.C; A.C
};
```

Here, we make an instance, `A`, of `aCFTemp`, and assert that the Fahrenheit and Celsius values of the temperature are the same value. The expressions in `aCFTemp` and `computeFEqualsC` are combined in order to solve for `A.C`.

Nested constraint patterns that define subobjects, and the constraint patterns constraining them, complement each other in a very nice way. Each subobject defines additional degrees of freedom, while a constraint on that subobject restricts those degrees of freedom. Within each object, the programmer's job is to manage the degrees of freedom in the object by adding subobjects and constraints restricting them. It is important to leave enough degrees of freedom available so that methods may temporarily further constrain the object to define new state. No degrees of freedom available implies that the object is fully constrained, and thus cannot change.

3.6.9 Scoping Rules

A subobject is visible within the lexical scope of the object that owns it. Generally, a subobject defined in an object `S` is visible:

- to `S`'s clients;

- to objects that use S as a prefix; and
- to any subobjects lexically enclosed in S.

The names of subobjects defined in an object body, as well as those defined in the object's prefixes, shadow the names of objects defined in any enclosing block. For example, if we have the constraint patterns

```
P: {
   a, b, c: aNumber;
};

Q: {
   a, c, d, e: aNumber;

   R: P {
      d: aNumber;
   };
};
```

the subobjects visible from inside R's body are

- P's a, b, and c;
- Q's e; and
- R's d.

Subobjects that are not intended to be used by external clients may be hidden by marking them with # (see section 2.2.1). Hiding a subobject limits its *outward visibility*. For example, if we would like to add a third hidden subobject to points that is the distance from the point to the origin, we could write

```
aPoint: {
   x: aNumber;
   y: aNumber;

   #distToOrigin: aNumber { sqrt(x^2 + y^2) };  -- Hidden subobject
   ...
   self
};
```

Here, aPoint defines the distToOrigin using x and y. Since distToOrigin is defined in the same scope as x and y, its body may use x and y directly, since they are lexically visible. distToOrigin is not visible to clients, but may be used within the definition of aPcint.

Normally all of an object's attributes are visible to derived objects that use it as a prefix. However, sometimes one would like to hide an object's attributes from both clients and other objects that refine it. This is called limiting an attribute's *downward visibility*. We protect `distToOrigin` from all access except internal to `aPoint` by marking it with `##`:

```
aPoint: {
   x: aNumber;
   y: aNumber;

   -- Protected attribute; not visible at all outside of aPoint.
   ##distToOrigin: aNumber { sqrt(x^2 + y^2) };
   ...
   self
};
```

The facilities for hiding an object's attributes from clients and from objects that use it are similar to the facilities in languages like C++. In Siri, all attributes of an object are visible (`public` in C++ terminology) and available to derived objects unless otherwise noted. In C++, attributes must explicitly be declared `public` in class definitions.

3.6.10 Accessing an Object's Evaluation Environment

Given an object, a client may access the object's attributes by evaluating expressions relative to an object's internal environment using an environment operator. There are two operators that provide this access: the *at-sign* operator (`@`), and the *dot* operator (`.`). We use the at-sign operator as an indexing operator when using an object as an array, and we use the dot operator to evaluate message expressions in the object's environment.

The At-Sign Operator

The at-sign operator provides a way to access subobjects of an object, just as subscripting operators provide a mechanism for accessing elements of arrays. We can create an object with named subobjects that allow us to use it in the same way as we would use an array; section 3.4.3 describes this *labeled repetition* feature.

The at-sign operator looks up subobjects in the object named as its left argument. To perform the lookup, the operator evaluates its right argument in the current environment, yielding a subobject name, which is then used to find the appropriate subobject. Numeric labels are marked with `@` to avoid confusing the objects they denote with the numeric value itself. Thus

```
myPoints@4      lotsOfRectangles@"t"
```

specifies the point named `@4` in `myPoints`, and the rectangle named `t` in `lotsOfRectangles` respectively.

The Dot Operator

The dot operator provides a way to enter the environment of an object for expression evaluation. The dot operator evaluates its right argument in the environment specified by its left argument, and in its simplest form is very similar to accessing record fields in a language like C or Pascal. For example, we can access `myPoint`'s `x` attribute by writing

```
myPoint.x
```

However, the dot operator is much more than a field or record selection operator: using the dot allows a client to enter an object's environment for evaluation of arbitrary expressions, similar to Pascal's `with` statement. We can compute a point's distance from the origin based on its `x` and `y` attributes by writing

```
myPoint.(sqrt(x^2+y^2))
```

The dot operator then sets the current environment to `myPoint`, evaluates the expression, binding `x` and `y` to the appropriate attributes of `myPoint`, and returns the result. This mechanism is the essence of message passing in Siri. Rather than choosing a procedure (method) based on a procedure name (message) and an object (receiver), the receiver is sent an entire expression to be evaluated within its own environment.[3] The dot operator handles its right argument specially: the argument is treated as if it is enclosed in brackets, thus delaying its evaluation until the environment specified by the left argument can be set.

The dot operator, along with the block construct, allows clients to use objects as remote computation environments. For example, we could use `aCFTemp` to compute a Celsius temperature given one in degrees Fahrenheit in the normal way, by setting up a constraint on `F` and requesting `C` as a value:

```
aCFTemp.F = 98.6; aCFTemp.C
```

The semicolon asserts that `aCFTemp.F = 98.6` is to be true, returning the value of `aCFTemp.C`. However, because the dot operator denotes `aCFTemp`'s environment, we could actually write the following to accomplish the same result, but more succinctly, as

```
aCFTemp.(F = 98.6; C)
```

Here we are using `aCFTemp`'s environment, including its subobjects and constraints, directly to compute the value of `C`.

---

[3]This is very similar to the evaluation mechanism of Smalltalk-72 [Kay93].

There is a major restriction on use of the dot operator. The dot operator may not modify the state of an object referenced by a shared attribute. Just as we may not modify attributes of a shared object, we may not evaluate expressions in a shared object's environment that would result in directly modifying its attributes. Modifying attributes can be done only via evaluating an object's methods.

3.7 Uses of the Constraint Pattern

Constraint patterns can be used for a variety of different purposes in a Siri program, depending on their labels and object descriptors, the context in which they are defined, the subpatterns that they have, and how they refer to other objects.

In this section, we discuss the different ways that patterns can be used to implement standard object-oriented constructs such as classes or prototypes, functions, instance variables, read-only attributes, side-effecting methods, and control structures. We also provide additional examples for extended constraint pattern concepts such as abstract classes, attributes, and methods.

We show that, by using a single extremely general and flexible abstraction mechanism, Siri provides a much simpler and more uniform way to express all these constructs, and makes possible generalizations that extend the functionality of traditional object-oriented programming.

3.7.1 Patterns as Classes and Modules

Any object created with a constraint pattern may be used as a prototype for instantiating other objects in its image. Typically, such patterns have simple labels that describe the kind of object that they create, such as `aRectangle` or `anInterval`. Patterns as classes normally define subpatterns for attributes, state, and method definitions that are used by each instance of the pattern.

A pattern to act as a class for fractional numbers with a numerator and a denominator might look like:

```
aFraction: aNumber {
   numerator, denominator: anInteger;

   reciprocal: aFraction   { ... };
   asFloat:    aFloat      { ... };
   ...
};
```

We could then use this already-defined pattern to instantiate fractions and use them in arithmetic expressions:

```
fractionProgram1: {
   a, b, c: aFraction;
   ...
};
```

Since classes usually define a set of operations on the objects that they define, one would expect messages to `aFraction` for addition, subtraction, and other arithmetic operations. However, constraint patterns allow a better mechanism for defining these operations through stand-alone patterns with the appropriate labels. In this way, both of the arguments to an operation such as addition may be examined in order to choose the matching pattern.

We could define then a complete package with fractions and their operations with a constraint pattern as follows:

```
Fractions: {
   aFraction: aNumber { --same as above-- };

   -- Fraction to Fraction operations
   "a'aFraction + b'aFraction": aNumber { --fraction addition-- };
   "a'aFraction - b'aFraction": aNumber { --fraction subtraction-- };
   "a'aFraction * b'aFraction": aNumber { --fraction multiplication-- };
   "a'aFraction / b'aFraction": aNumber { --fraction division-- };

   -- Fraction to Integer operations
   "a'aFraction + b'anInteger": aNumber { --fraction or integer result-- };
   "a'anInteger + b'aFraction": aNumber { --fraction or integer result-- };
   ...
};
```

Then we could use this package in a program by using it as a prefix:

```
fractionProgram2: Fractions {
   a, b, c: aFraction;
   a + b = c;

   ...more code creating and manipulating fractions...
};
```

A fraction package that includes both the fraction object definition, and patterns that define the fraction's operations with other kinds of objects, is similar to a class; however, it provides a modular mechanism for encapsulating, and using, stand-alone patterns for evaluating expressions. See Appendix A for more examples of this kind.

3.7.2 Patterns as Functions

Patterns with parameterized labels can be used to define functions, where the parameters in the pattern label serve as the function's arguments. Functions may have a variety of different syntaxes, depending on the label's operators; standard functional syntax is given here. Note that in Siri, arguments to functions are always annotated with their types since they are simply label parameters, thus, a single function name can be overloaded by a set of distinct constraint patterns, each matching a particular set of types for parameters.

For example, one could define the *greatest common denominator* function as follows:

```
"gcd(a'anInteger, b'anInteger)": anInteger {
   if (b = 0) then a else gcd(b, a \\ b)
};
```

where `a \\ b` denotes the remainder when `a` is divided by `b`.

We can take advantage of nested constraint patterns for functions that have internal cases or utility routines. For example, here is a function that returns the nth Fibonacci number:

```
"fibonacci(n'aNumber)": {
   -- Base cases. All utility functions hidden from clients.
   #"fib 0":            aNumber  { 1 };               -- Case for n=0
   #"fib 1":            aNumber  { 1 };               -- Case for n=1
   #"fib n'aNumber":    aNumber  { fib2(1, 1, n-2) }; -- Case for n>=2

   -- General case
   #"fib2(a'anInteger, b'anInteger, n'anInteger)": aNumber {
      if n <= 0 then b else fib2(b, a+b, n-1);
   };

   -- The value of fibonacci(n'aNumber)
   fib n
};
```

In this example, the base cases are matched directly by an overloaded utility function, `fib`, for values of `n=0`, `n=1`, and other values of `n`; the general case calls a tail recursive routine, `fib2`, that computes the Fibonacci function by maintaining the last two elements in the sequence and the current index. The value of the entire constraint pattern is the final expression, `fib n`. A single constraint pattern thus may package local functions together in order to group them together for local use, and to hide them from clients.

3.7.3 Patterns as Instance Variables

Instance variables are simply storage locations for values that are manipulated directly by methods. Constraint patterns with simple labels, a prefix, and no body, when defined inside of another constraint pattern, can play the role of instance variables. For example, an `Interval` object that describes an integer range is defined as follows:

```
anInterval: {
   #beginning: anInteger;
   #ending:    anInteger;
   #increment: anInteger;

   ...no expressions relating beginning, ending, or increment...
};
```

Here, the subobjects `beginning`, `ending`, and `increment` play the roles of instance variables, since there are no constraints on them limiting their values. Each subobject is also hidden from clients by being marked with a `#`. Because they are hidden, one would have to provide other unhidden attributes that are functions of the hidden ones for client use. In addition, the only way to change subobjects' values would be through assignment or constraint operations inside of a method definition.

3.7.4 Patterns as Object Attributes

A Siri program generally will not have stored state instance variables. Instead, it will have attributes that serve the same purpose, but that also may be viewed by clients and have constraints on their state. Attributes are typically constraint patterns with simple labels that describe a particular characteristic of the object in which they are defined.

For example, we may define a geometric vector in a plane as follows:

```
aVector: {
   rho:     aNumber;                    -- the vector's length
   theta:   aNumber;                    -- the vector's angle
   origin:  aPoint;                     -- The vector's origin

   dx: aNumber { rho * cos theta };     -- change in x from origin
   dy: aNumber { rho * sin theta };     -- change in y from origin

   endPoint: aPoint { x = origin.x + dx; y = origin.y + dy; self };
};
```

In this example, all of the attributes are visible externally; some are stored directly as values, and some are computed in terms of the others.

3.7.5 Patterns as Methods that Modify State

To maintain encapsulation, a Siri object defines an interface to methods for internal state modification. A method is a special kind of constraint pattern that enforces constraints temporarily only during method evaluation. The method uses these constraints to direct the constraint satisfaction process to a unique solution for the object's attribute values.

Method patterns are defined using a constraint pattern with the prefix `aMethod`. `aMethod` provides several new operators for use within the method body:

- a `fixed` operator, that constrains an attribute to stay the same during the method pattern evaluation;
- a `previous` operator that provides the previous value of an attribute;
- an `is` binding operator for directly assigning values to attributes.

The method pattern allows the programmer to set up a context for the system to compute new values for the unfixed and unbound objects. By fixing the values of some of the objects in the constraint pattern, the programmer reduces the degrees of freedom enough that the system can uniquely solve for the remaining values using the newly bound objects. Because the system knows what values are desired, what values are fixed, and what values may be changed, the equation solving patterns can partially evaluate the method pattern body while leaving the parameters unknown. This leaves a simplified set of equations that can be evaluated quickly when the method pattern is invoked. Special constraint patterns also can be written to compile the method pattern to a sequence of machine instructions for execution (see [Leler88]).

The following example is an expansion on the previous example of `aRectangle`, with several new method patterns for initialization, moving, and resizing:

```
aRectangle: (
   -- Attributes
   -- These define the coordinates of each edge in x or y.

   top, left, bottom, right: aNumber;

   width:          aNumber   { self = right-left; self };
   height:         aNumber   { self = bottom-top; self };
   topLeft:        aPoint    { x = left; y = top; self };
   bottomRight:    aPoint    { x = right; y = bottom; self };
   center:         aPoint    { x = left + width/2; y = top + height/2; self };

   --Method Patterns

   initially: aMethod {
      width = 100; height = 50;
      center = point (400, 300);
   };

   "moveTo newCenter'aPoint": aMethod {
      width, height fixed;
      center = newCenter;
   };

   "moveBy delta'aPoint": aMethod {
      width, height fixed;
      center = previous center + delta;
   };

   "resize newCorner'aPoint": aMethod {
      topLeft fixed;
      bottomRight = newCorner;
   };

   self
};
```

Figure 3-1. An extended implementation of `aRectangle`.

In this example, the rectangle is initialized when first created by the `initially` method pattern, which is automatically invoked by the system. By temporarily constraining the `width`, `height`, and `center`, the system can solve for the initial object state and the values for `top`, `left`, `bottom`, and `right`. In the other method patterns, attributes are fixed by constraining them to their current values (`topLeft fixed`), while other attributes are constrained to new values (`bottomRight = newCorner`). Evaluating these patterns with parameters binds `top`, `left`, `bottom`, and `right` to new values determined by the equation solver.

3.7.6 Patterns as Abstract Classes, Attributes, and Methods

We have seen how to define class-like structures, attributes, and methods using constraint patterns. We can also define abstract versions of these constructs that omit implementation details. Abstract classes and attributes instead define only the types of arguments and values, and the intent of methods, coded purely in the form of constraints on attributes.

For example, we could define an abstract rectangle object as

```
anAbstractRectangle: {
   top, left, bottom, right, width, height: aNumber;
   center: aPoint;

   "moveTo newCenter'aPoint": aMethod {
      width, height fixed;
      center = newCenter; };
   ...
};
```

This abstract rectangle does not describe at all the rectangle's implementation, or how its attributes are related to each other; it describes only its interface, through the given abstract attributes. Using this abstract rectangle we can provide a rectangle implementation that considers the `top`, `left`, `bottom`, and `right` coordinates as basic attributes, and describes the other attributes in terms of them:

```
anEdgeRectangle: anAbstractRectangle {
   width:   { self = right - left; self };
   height:  { self = bottom - top; self };
   center:  { x = left + width/2; y = top + height/2; self };
};
```

This implementation defines constraints for `width` and `height` in terms of the coordinates, and defines the `center` point in terms of the coordinates and the dimensions. No prefixes are needed on the attributes of `anEdgeRectangle` because they are automatically prefixed by their namesakes in `anAbstractRectangle` through the recursive prefixing process. Thus, `width` and `height` are automatically prefixed by `aNumber`, and `center` is prefixed by `aPoint`.

We can provide instead an implementation that considers the `width`, `height`, and the `center` point as basic attributes, and defines `left`, `right`, `top`, and `bottom` in those terms:

```
aCenterRectangle: anAbstractRectangle {
   left:    { self = center.x - width/2; self };
   right:   { self = left + width; self };
   top:     { self = center.y - height/2; self };
   bottom:  { self = top + height; self };
};
```

By not redefining `width`, `height`, and `center`, we inherit directly the definitions for these attributes from `anAbstractRectangle`.

Notice that in both cases, the abstract method `moveTo` is not redefined; it is interpreted in each concrete pattern differently, depending on how `width`, `height`, and `center` are defined.

By choosing different implementations of `aRectangle`, we can control how we structure the object's attributes. In section 5.3.3 we describe how to hint to the system that we want to store some of the attributes as state, and compute others as functions of the state. Abstract patterns and methods make it possible to leave these decisions to derived objects, while still providing a basic framework with constraints and method definitions.

3.7.7 Patterns as Control Structures

So far we have seen only declarative code with no sequencing or looping. We may define control structures using constraint patterns as well, by using blocks to control the order of evaluation. In this section we present three basic control structures: a sequence assertion; a sequenced `if` statement; and a `while` loop.

Sequencing

The expression `A ! B` denotes the evaluation of `A` followed by `B`. The left argument must evaluate to `true`, just as required by the semicolon assertion operator. So that `B`'s evaluation is delayed, it must be enclosed in square brackets; for symmetry's sake, we will require both `A` and `B` to be enclosed in square brackets. So an expression

```
[ eraseBackground ] ! [ drawWindow ] ! [ drawContents ]
```

will first erase the background, then draw the window, and finally draw the window's contents. The constraint pattern defining sequencing is as follows:

```
"[ s1'expr ] ! [ s2'expr ]": aMethod {
   "true do [s'expr]": { s };
   s1 do [s2]
};
```

This pattern strips off the brackets from the expression `s1` and evaluates it within the environment of the pattern. By enclosing the expression `s2` in blocks, its evaluation is delayed. On instantiation of the pattern, `s1` is immediately evaluated in the expression `s1 then [s2]`, which rewrites to `true then [s2]`. Then, the expression `true then [s]` can be matched, evaluating `s2`.

However, because sequencing is such a common construct, we would like to avoid having to write brackets around each of our statements; instead of writing

```
[ eraseBackground ] ! [ drawWindow ] ! [ drawContents ]
```

we would prefer to write

```
eraseBackground ! drawWindow ! drawContents
```

We thus handle the sequencing operator specially, in that it by itself indicates that its arguments are not evaluated. So our sequencing definition is then simply

```
"s1'expr ! s2'expr": aMethod {
   "true do [s'expr]": { s };
   s1 do [s2]
};
```

and we can write our imperative code without using square brackets.

Sequenced If Then Else

A sequenced `if` statement is useful in that often the consequence of an `if` should not be evaluated unless the test is `true`. The method pattern for sequenced `if` is as follows:

```
"if t'expr thenDo s1'expr elseDo s2'expr": aMethod {
   "true  doTrue [s'expr] doFalse [ignore'expr]":  { s };
   "false doTrue [ignore'expr] doFalse [s'expr]":  { s };
   t doTrue [s1] doFalse [s2]
};
```

Like the sequencing operator, the `if_then_else` operator does not allow its arguments to be evaluated. The value of a sequenced `if` expression is simply `s1` if `t` is `true`, and `s2` otherwise.

While

Using the sequence assertion and sequenced `if`, we can easily write a `while` loop as follows:

```
"while t'expr do b'expr": aMethod {
   loop: { if t thenDo (b ! loop) elseDo true };
   loop
};
```

The `while_do_` operator as well does not allow its arguments to be evaluated. The loop is begun and a tail-recursive call to `loop` is made as long as the value of the expression `t` evaluates to `true`.

Blocks and Inner

As we have seen, blocks defined by square brackets are used to hold expressions that are to be stored and evaluated at a later time. Blocks are especially useful in allowing us to define control structures using constraint patterns.

We may also use blocks to parameterize a routine by giving it a lambda-like structure which can be repeatedly evaluated; for example, a sorting routine can be passed a comparison block to compare two elements in the sorting process. In Smalltalk, for example, a sort routine can be parameterized by a `sortBlock` that is evaluated to determine whether a particular element should be sorted before another.

However, there is a better way to parameterize objects in Siri: the `inner` mechanism is used in constraint patterns to specify a refinement for an object or routine. Instead of providing a block to be evaluated in a sorting routine, the sorting routine can be defined abstractly, using the keyword `inner` to denote the point at which the refinement's behavior is invoked. By prefixing code that performs a comparison between two entries in a list with the sorting code, we can define a combined object with the desired sorting behavior. For example, if we have a sorting routine

```
"quickSort s'anArray": anArray {
   --Temporary objects used during sorting
   #elt1, #elt2: anObject;
   ...
   if inner then ...elt1 before elt2... else ...elt1 after elt2...;
};
```

we can specialize our own sorting routine to sort using the < operator:

```
"mySort s'anArray": "quickSort s'anArray" { elt1 < elt2 };
```

Here, `inner` is used to allow `mySort` to specialize `quickSort` by providing an element comparison. The objects `elt1` and `elt2` are directly inherited from `quickSort` to be used in `mySort` to specify the comparison.

List Comprehension

Another convenient control mechanism made possible by `inner` is known as list comprehension. List comprehension allows the evaluation of a particular expression over a list of objects. List comprehension is handled in languages like Smalltalk through the use of blocks; for example, to compute the sum of a list of integers in Smalltalk, one can write

```
(1 to: 10) do: [:i | sum <- sum + i. ]
```

This sends the message `do:` to the list of integers, which iterates through the list evaluating the block, which is passed unevaluated to `do:`. Blocks in Smalltalk are like lambda expressions in that they accept a parameter whose value is to be used in the block code.

With constraint patterns, however, we would like to expand the block into a set of persistent expressions, which are then evaluated as a group later by the solver. For example, one might like to align the bottoms of a group of rectangles to a particular y location. Using constraint patterns we can write

```
(rect1, rect2, rect3) map { each.bottom = alignLocation; ... };
```

where we define mapping as

```
"l'aList map": { each: anObject; each = l.head; inner; l.tail map inner };
```

The result of the evaluation of this constraint pattern is an object that is used as a prefix to another object. Rather than provide a block which is evaluated explicitly with parameter passing as is done in Smalltalk, we map constraint patterns by using the `inner` mechanism to provide a namespace for the temporary object `each`, and to expand the `inner` expressions inline with the object definition[4]. Use of list comprehension is not limited to expanding constraint expressions: imperative code may also be evaluated using map with the sequencing operator in methods.

3.8 Limitations of Siri and the Constraint Pattern Abstraction

Siri and the constraint pattern abstraction as presented here do have some limitations, due to the model, the abstraction, and the current implementation.

[4]The ellipses are a syntactic placeholder to allow parsing of incomplete expressions.

3.8.1 Separate Constraints and Messages

Constraints and messages are used for two distinct purposes in Siri's constraint patterns. Objects that define internal constraints restrict their subobjects to communicate only through constraints; they may not send messages to each other. This is due to the fact that constraint satisfaction in Siri requires, to a certain degree, internal knowledge of the implementation of each of the objects in a constraint network. Message passing, on the other hand, requires that there be no knowledge of an object's implementation details, but that each object is responsible for providing its own methods for carrying out the intent of messages sent to it. The constrained objects model's restriction to a two level hierarchy, constraints inside of encapsulated objects and messages outside, provides a simple separation of the two distinct concepts of constraints and message passing.

One might prefer a more uniform system in which constraints and messages can be freely intermixed at all levels. The ability to either constrain an object or send a message to it would require a different definition of encapsulation, since an object may have its state changed without going through its interface, thus giving up control of its own state consistency.

3.8.2 Limited Constraint Domain

For constraint patterns to be executable, we need to define a particular domain over which the constraints will be solved. Siri currently limits constraint patterns to solving simple linear and slightly non-linear algebraic constraints over the real numbers. The constraint patterns that implement the solver use objects to create and manipulate ordered linear combinations (see section 4.2.2); with these, variables may be solved for in terms of other variables and objects. We chose this mechanism for its simplicity and applicability to a wide variety of typical object-oriented programming tasks; there are, of course, problem domains for which this is inadequate.

There are two ways to handle the situation in which the constraint solver is not powerful enough to solve a given problem directly. The first approach is to program the solution directly, just as one would in a non-constraint-based language: with objects, message passing, and implicit constraints. The second approach is to extend the solver by writing rules to satisfy constraints in the new domain. While the first approach may be simpler, since it is a special-purpose solution to a problem in a given domain, the second approach is more general since the more-powerful solver may then be used for a wide variety of problem formulations in the domain. See Chapter 5 for details on extended solvers.

3.8.3 No Constraints on Shared Attributes

A constraint pattern with a shared attribute referencing an external object may not have constraints on that attribute. This restriction was placed in order to close a loophole that would allow changing of an object's internal state without going through its interface. If constraint patterns did allow constraints on a shared object, the shared object could arbitrarily cause the

state to change of any object that references it.

Instead of allowing direct constraint-controlled changing of state by a shared object, constraint patterns provide the `changed` message mechanism that informs an object that a particular one of its shared attributes has changed state. Upon receipt of a `changed` message, the object has complete freedom to update its own state as it sees fit, by accessing the shared object's attributes and temporarily constraining its state based on that information.

3.8.4 No First-Class Constraint Patterns

The current semantics of constraint patterns do not allow for their manipulation as first-class objects: a constraint pattern itself may not be stored, passed as an argument, or arbitrarily applied to an expression or list of arguments.

Constraint patterns with parameterized labels are not functions or procedures; they are not directly applied to a list of arguments. Instead, the system uses constraint patterns defined in a particular scope to modify and eventually evaluate an expression. One can think of a parameterized constraint pattern as an active method that applies itself to whatever subexpressions it matches in its scope.

3.9 Advantages of Siri and the Constraint Pattern Abstraction

Although Siri and its constraint pattern abstraction as defined in this thesis do have some limitations, Siri is still a complete object-oriented language. There are several benefits to a language based on constraint patterns: a single abstraction mechanism for the entire language; fully encapsulated objects; and more expressive method coding.

3.9.1 A Single Abstraction Mechanism

We use the single constraint pattern abstraction to define individual objects, prototypes, methods, attributes, abstract prototypes and methods, and control structures. Objects defined by constraint patterns use prefixing to inherit characteristics from other objects, and nesting to define owned objects.

Prefixing

The prefixing mechanism is applicable to many situations. While prefixing can be considered simply to be an inheritance mechanism, it can be applied to more than just class-like objects to which traditional inheritance is often limited.

- Attribute patterns may be prefixed by other attributes, refining their values;
- Method patterns may be prefixed by other methods, combining their effects.

- Objects may be prefixed by one or more existing objects, inheriting their attributes, methods, and constraints.

Nesting

Because constraint patterns may be nested arbitrarily deeply, object definitions may be also nested and used locally within a pattern. Objects thus may act as modules that contain other object definitions, and can be included for use simply by prefixing. In addition, a prototype object acting as a class that is used only in a single object may be hidden within that object; the ability to define any kind of object at any level can significantly reduce clutter in the system namespace.

For more details and examples of programming in a single-abstraction language with prefixing and nesting, see the BETA manual [Madsen89].

3.9.2 Fully Encapsulated Objects

Siri's constraint patterns provide encapsulated objects that are fully responsible for their own internal state. Client objects may not change the state of another object without going through the its message interface.

This is a very important feature. In most constraint languages, objects have complete access to other objects via constraints on their parts. In Siri, only within an encapsulated object do subobjects have access to each other's state. Encapsulated objects in Siri protect their internal state from others by restricting access to a defined interface, and maintain their internal state consistent via constraints.

3.9.3 Expressive Method Coding

Method code is more expressive, more abstract and simpler in Siri than in traditional imperative object-oriented languages such as C++. In Siri methods, we use attributes as abstract, measurable characteristics of an object for expressing how that object should change: which attributes should be modified and which should stay the same.

By coding methods with respect to an object's attributes, we can encode more specifically the intent of the method. In imperative method code, it is up to the programmer to both implement an implicit intent in all methods, and to understand it by deciphering method code to unveil the previous programmer's assumptions.

Chapter 4
A Siri Implementation

In this chapter we discuss the implementation details of a prototype of Siri that implements the constrained objects model using constraint patterns. There are three aspects to execution of a Siri program, each of which is implemented using the ATR+ rewriting mechanism. The first aspect, described in section 4.1, is the reduction of a constraint pattern to its corresponding Siri object. The second aspect, described in section 4.2, is the constraint satisfaction process that occurs within objects; this is accomplished simply by applying the reduction process to each object's algebraic constraint expressions. The third aspect, described in section 4.3, is the process of modifying objects by method evaluation, again using the reduction process to perform constraint satisfaction. Finally, the last section, section 4.4, outlines the in-memory structure of Siri objects that are the result of constraint pattern reduction, and describes the intrinsic objects defined by the prototype Siri system.

4.1 Evaluating a Constraint Pattern

Siri evaluates constraint patterns through ATR+, a reduction process that transforms constraint patterns into in-memory structures. ATR+ performs augmented term rewriting of expressions, creates object state, and merges sets of constraint pattern definitions into a single combined object. Since constraint patterns define objects, rather than rules, the reduction process uses augmented term rewriting both to create and manipulate objects and to transform subexpressions.

The ATR+ augmented term rewriting mechanism as described here is a general-purpose computation mechanism. While ATR+ itself does not perform constraint satisfaction, Siri can do so with appropriate constraint pattern definitions. Siri uses constraint satisfaction to simplify equations in object definitions, solve for attribute values, and compile satisfaction methods for object modification.

ATR+ reduces a constraint pattern by rewriting a *subject expression* that consists of expressions defined in both the constraint pattern's prefixes and body. This process creates subobjects defined by nested constraint patterns, and rewrites the pattern's original expression to a new, equivalent one using other definitions that are visible in the pattern's scope. The process is complete when there are no more reductions that can be made on the subject expression. The result of reducing a constraint pattern is:

- an object that represents the result of the constraint pattern's evaluation;
- internal subobjects that represent the object's attributes; and

- a set of expressions that remain, called the *residual*. The residual is the object's value, and may include constraints on the object.

Object reduction consists of several stages. First, a parser transforms the constraint pattern into an expression tree that can be manipulated by the ATR+ engine. Then, the ATR+ engine repeatedly

- matches object labels to reducible expressions in the pattern;
- instantiates objects that match subexpressions; and
- replaces the subexpressions with the corresponding objects' residuals.

When no labels match subexpressions in the pattern, the reduction process ends.

Sections 4.1.1 and 4.1.2 describes constraint pattern parsing and matching reducible expressions. Section 4.1.3 provides details on the constraint pattern residual. Finally, sections 4.1.4 describes the object instantiation processes.

4.1.1 Parsing a Constraint Pattern

Siri uses a simple operator-precedence parser to transform the textual constraint pattern definition into a syntax tree. Special constraint patterns define operators (see 3.4.2) that are used to parse other constraint patterns; some basic operators, such as the colon, braces, and type quote, are predefined in the interpreter.

Unary and binary operators are handled exactly as specified in normal operator precedence parsers. However, we can also define mixfix operators that specify a group of operator keywords in a single definition. For example, the mixfix interval operator introduced in Chapter 3 as defined by

```
"from_to_by_":    { %%precedence is 5; %%opType is anExprOp; self };
```

produces a mixfix operator with three keywords, `from`, `to`, and `by`; specifies the type of the expression to be `anExprOp`; and indicates that the precedence of the operator as a whole (and each keyword) is `5`. Each keyword is considered an operator as far as the parser is concerned. Depending on the form, the mixfix operator assigns precedence, arity, and associativity to the keywords in order to parse correctly expressions that use the mixfix operator.
In the above example, the mixfix operator will assign precedence, arity, and associativity to the keywords as follows:

keyword	arity	associativity	precedence
from	unary	none	5
to	binary	left	5
by	binary	left	5

Using these keywords, the parser translates the expression into a syntax tree, and then collapses the three keywords into a single operator node with three children. For example, the parser parses the expression

```
from 10 to 20 by 2
```

into the following:

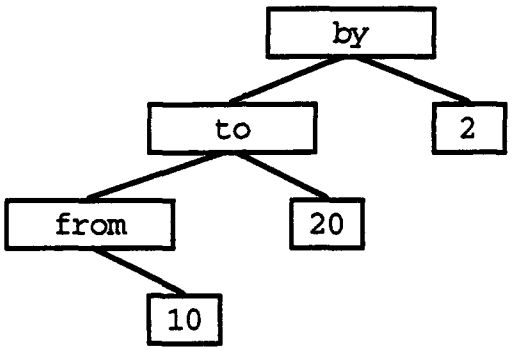


Figure 4-1. A syntax tree for the expression `"from 10 to 20 by 2"`

and then flattens the tree to the following:

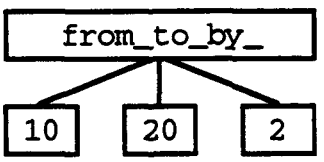


Figure 4-2. A flattened syntax tree for the expression `"from 10 to 20 by 2"`.

Keywords defined in a multi-part operator, as above, may appear in more than one definition as long as they are used consistently. Specifically, they must always have the same precedence, arity, and associativity in each definition; if they do not, or if they overlap in such a way that the multi-part operator can be parsed ambiguously, it is considered an error.

Some operators are handled specially in order to perform a certain amount of processing during the parsing operation. The colon operator, `_:_`, instantiates new objects in the current environment; the interpreter creates objects immediately at parse time so that, for example, primitives and labels may be defined for subsequent use. The interpreter handles the typequote operator, `_'_`, at parse time for typing of parameter variables. Parentheses `(_)` are a parse-time operator used simply for grouping, and do not have a runtime interpretation.

Finally, the parser uses the braces operator, `{_}`, to define a new object environment. When the parser matches a left brace, the interpreter pushes the old environment onto an environment stack, and creates a new object to be the current environment. Subsequent object instantiations then occur within the new environment. When the parser matches a right brace, the interpreter pops the previous environment off of the stack, sets it to the current environment, and pushes it onto the operand stack for further parsing. This process allows the parser to handle nested constraint pattern definitions in a single pass.

Once the parser transforms the expressions into a syntax tree, the matcher and instantiator begin matching reducible expressions to instantiate into objects.

4.1.2 Matching Reducible Expressions

The matching phase of reduction matches labels against reducible subexpressions, or *redexes*, in the current subject expression. Each parameterized label in Siri is an expression with variables. In the current implementation of Siri, each operator maintains a list of triples of `(label, object, environment)` for which that operator is the root operator of each label in the list. For example, the multiplication operator `*` will maintain a list of objects defining labels such as

```
a'aNumber * b'aNumber
a'aNumber * b'aNumber + c'aNumber
a'aNumber * b'aNumber / c'aNumber
```

since `*` is the root operator of each of the labels. This is done to improve performance: given a subexpression, the matcher only needs to try the labels listed by the subexpression's root operator that are visible from the current evaluation environment.

Given an expression in an environment, the matching process proceeds as follows:

```
Match(expr, env): {
   labels = labels of the expression's root operator.
   for each label in labels visible from env do {
      labelArgs = arguments of this label.
      exprArgs = arguments of the expression's root operator.
      for each labelArg in labelArgs & exprArg in exprArgs in step do {

         if isAnExpression(labelArg) & isAnExpression(exprArg) then
            recursively match on parts of labelArg and exprArg

         else if isAVariable(labelArg) then
            if isAnExpression(exprArg) and
               variableType(labelArg) = expressionType(exprArg) then MATCH
            else if isAnObject(exprArg) and
               variableType(labelArg) = type(exprArg) then MATCH

         else if isAnObject(exprArg) & isAnObject(labelArg) then
            if object identifiers equal then MATCH
      }.
   }.

   if there is no match in any label, then
      recurse down the tree, breadth first, matching subexpressions using
      the same environment.
}.
```

Figure 4-3. The `Match` algorithm.

Once a match occurs, the matcher binds the variables in the label to their corresponding terms in the redex. The instantiator uses these bindings in the next phase to create the parameterized object.

A much faster implementation of matching is possible: we can take advantage of our previous restriction that labels must obey strict left-sequentiality (see 3.2.1) to allow the matcher to use a fast string-based pattern matching algorithm. See Chapter 5 and [Leler88] for details.

The labels of constraint patterns match redexes in a subject expression through a mechanism that matches symbols to symbols, operators to operators, and variables to expressions and objects. Because the ATR+ matching mechanism examines the entire expression, it can match polymorphic expressions that include variables of multiple types. Constraint patterns such as

```
"a'anInteger+b'aFloat": aFloat          {...};
"a'aRational+b'anInteger": aRational    {...};
"a'aNumber+b'aNumber": aNumber          {...};
```

use the knowledge of the types of both `a` and `b` to determine which constraint pattern is being specified. In both cases, the operator `+` is a binary, left associative operator with precedence; the differences in the three label expressions are simply the types of objects which surround the operator, and thus all operators are polymorphic. Overloaded operator matching avoids the unnatural and asymmetrical interpretation of `"a+b"` as "sending the message `+` to `a` with parameter `b`" that many object-oriented languages require.

The ATR+ matching mechanism orders expressions by specificity. More specific types match before expressions with less-specific types, thus `"a'integer + b'float"` would match before `"a'number + b'number"`. To order expressions that have equally-specific variables, we use a tie-breaking mechanism: in Siri, the leftmost type in an expression is considered the most significant. Thus, `"a'float + b'number"` is more specific than `"a'number + b'float"`.

4.1.3 The Residual

An object's *residual* is the expression that remains after the reduction of its constraint pattern in its evaluation environment. This expression defines the object's value, and it replaces the redex during the augmented term rewriting process. If the object cannot be reduced to a single value, the residual is an expression that describes as-yet-unresolved relationships between the object's attributes.

While an object's attributes may have been related to each other with complex expressions, the rewriting process reduces these expressions to simpler ones by inlining other patterns within the scope of the object, resulting in a more compact residual. For example, the body of

```
"a'aNumber wildOp b'aNumber": {
   c: aNumber { self = a - 2*b; self };
   d: aNumber { self = b - a - c; self };

   c + d - a + 2*b = 14;
};
```

has a residual, after reduction, of

```
2*a - 3*b + 14 = 0;
```

Evaluation of the constraint pattern results in the creation of two numbers, `c` and `d`, in the scope of the object `"a'aNumber wildOp b'aNumber"`, and the reduction of the equation relating them. The equation solving patterns reduce the equations to the simpler ordered linear combination `2*a - 3*b + 14 = 0;`, which is the residual. The residual thus expresses the constraints

between the parameters a and b; the subobjects c and d have their own residuals, a - 2*b, and -2*a + 3*b respectively, which relate those objects to values in their environment. Subsequently matching the constraint pattern to the expression

```
10 wildOp N
```

where N is aNumber, results in the following expression, after some algebra:

```
20 -3*N + 14 = 0;
```

Here, the residual does not refer to the subobjects c and d since they had been used to find the solution to the original body expression c + d - a + 2*b = 14;. On instantiation, Siri simply makes a copy of the residual and replaces the variables with their parameter values; any references to subobjects defined in the object itself remain.

Once an object has been instantiated, its residual replaces the matched redex. Since the residual is an expression that can reference attributes within the object, it acts as a proxy for that object in the expression, ensuring that the object's attributes will always retain the asserted relationships.

Residuals can be considered to be extensions of values returned by procedures. Instead of returning a single value, an object, when instantiated, can replace the expression that it matched with a new set of expressions that relate values to each other. For example, given

```
a, b: aNumber;
2*(a+b) = 14;
```

instantiating the object defined by the constraint pattern

```
"2*(r'aNumber+s'aNumber)": aNumber { self = 2*r + 2*s; self };
```

would result in the expression

```
2*a+2*b = 14;
```

Thus, the value of the instantiated object is an expression relating the objects a and b.

Part-Whole and Encapsulation

ATR+ can gain access to subobject definitions in nested scopes for rewriting purposes. The residual of a subobject is, in some sense, a piece of the definition of the object that is exported to other scopes for evaluation. Within an encapsulated object, it makes optimization possible by removing the interface barrier between the objects, exposing their low level constraints which the equation solver can use. For example, if we have

```
easyMath: {
   a: aNumber;
   b: aNumber { self = 10 + a; self };
   a + b = 16;
   a, b
};
```

we can use the body of b for rewriting purposes. Since b's own residual after evaluation is `10 + a`, the equation solving patterns can use it in the expression `a + b = 16` to solve for a initially, and then b.

By substituting b's residual for b in `a + b = 16`, and solving the resulting expression, we find that `a = 3`. We then substitute this value back into b's residual, and determine that `b = 13`. In this way, Siri binds a and b to their corresponding values in the object, and the residual of `easyMath` simply becomes the list `3, 13`.

4.1.4 Instantiating Objects

ATR+ instantiates objects in two different ways during constraint pattern evaluation. Reducible expressions that match an object's label create an instance of that object in the current environment by *parameterized instantiation*. Labeled constraint patterns create permanent objects that become named parts of an object's environment by *prefixed instantiation*.

Parameterized Instantiation

Parameterized instantiation occurs when an object's label matches a redex. ATR+'s parameterized instantiation corresponds to augmented term rewriting's match-and-replace step. As part of the matching process, the variables in an object's label are bound to the appropriate terms in the matched redex, and used later in the instantiation process to produce an object parameterized by those values. For example, given the constraint pattern

```
"a'aNumber pt b'aNumber": aPoint { x = a; y = b; self };
```

we can create a new point with the expression

```
20 pt 30
```

which results in the creation of the point `(20, 30)`.

Parameterized instantiation is performed by the following steps:

- First, Siri clones the object whose label matched the expression;

- Siri creates a copy of the residual expression, replacing variables by the label's bound values; and finally,

- Siri continues to reduce the residual in the evaluation environment.

Once the object has been instantiated and its residual reduced, Siri replaces the matched expression with the new object's residual. More formally, we describe the procedure of instantiating an object with parameters as follows:

```
PInstantiate(A, params): {
   Create new B with length = length(A).

   Clone(A): {
      For each field f in A do
         if f is not the residual field then
            if immediate(f) or reference(f) or shared(f) then B.f = A.f.
            if owned(f) then B.f = PInstantiate(A.f, params).
         else
            B.f InstantiateResidual(A.f, params).
   }.

   Reduce(B.residual, B).
   Return B
}.
```

Figure 4-4. The `PInstantiate` algorithm.

where we define `InstantiateResidual` and `Reduce` as the following:

```
InstantiateResidual(expr, params):
   create newExpr as a copy of expr (copy variables directly, not values)
   for each variable Var in params with value Val
      replace all occurrences of Var in newExpr with Val.
   return newExpr

Reduce(expr, env):
   perform reductions on expr in env until no more reductions are possible.
   return expr
```

Figure 4-5. The `InstantiateResidual` and `Reduce` algorithms.

In the previous example, we bind the label variables a and b to 20 and 30 respectively. Siri then creates a new instance of aPoint and copies the residual of the matching instance (e.g., the object with label "a'aNumber pt b'aNumber"). The residual is simply

```
x = a; y = b; self
```

and so InstantiateResidual replaces a with 20 and b with 30, returning

```
x = 20; y = 30; self
```

Continued reduction of the residual results in binding x and y to the point's coordinate attributes. After binding these values, Siri rewrites them to true and the final residual becomes simply self, returning the newly-created instance of aPoint to replace the matched subexpression 20 pt 30.

Prefixed Instantiation

Prefixed instantiation occurs when Siri evaluates a constraint pattern with a prefix. Prefixed instantiation in Siri's ATR+ is an extension of ATR's labeling process for object creation. Each of the following expressions results in a prefixed instantiation in Siri, though only the first is expressible in Bertrand:

```
p: aPoint;                                  -- Creates an unconstrained point
theOrigin: aPoint { x = 0; y = 0; };        -- Creates the point (0, 0)

aSquare: aRectangle, SidesEqual;            -- Creates a square inheriting from
                                            -- two different prefixes

mySquare1: aSquare { width = 20; };         -- Creates a square with side = 20

mySquare: aRectangle, SidesEqual { width = 20; };  -- Also creates a square
                                                   -- with side = 20
```

Prefixed instantiation is different from parameterized instantiation in that prefixed instantiation defines a new, persistent object in the current environment based on an object descriptor, while parameterized instantiation creates a temporary object based on a reducible expression. In addition, parameterized instantiation requires only the cloning of a single object and the instantiation of its residual with parameters; prefixed instantiation is more involved, requiring the merging of multiple objects and their residuals.

The process of merging multiple objects and their residuals is called PrefixMerge. PrefixMerge consists of two basic functions: cloning and merging of the object fields to create a new object, and merging of the residuals of each prefix to create a combined residual.

In chapter 2, we discussed the syntactic interpretation of prefixing that recursively prefixed subobjects with their namesakes in the prefix. `PrefixMerge` is the process that performs this merging on objects that have been at least partially reduced, and exist as in-memory objects with state and reduced expressions.

Specifically, given a constraint pattern of the form

```
pObject: PrefixList { pBody };
```

we perform `PrefixMerge` to merge together the object's attributes and their residuals (Figure 4-7). `PrefixMerge` calls `ResidualMerge` to conjoin the residuals of the body and of each of the prefixes, handling special cases as necessary.

```
ResidualMerge(prefixResidual, baseResidual): {
   addResidual = copy of prefixResidual.
   merge addResidual with baseResidual through conjunction.
   Merge point is indicated by inner.
   Handle return result cases as given below.
   return modified baseResidual.
}.
```

Figure 4-6. The `ResidualMerge` algorithm.

```
PrefixMerge(pObject, PrefixList, pBody): {
   residual = copy of pBody;

   for each prefix p in PrefixList do {
      if p has indexed fields then
         if indexedObject = NIL
            then indexedObject = p.
         else
            error: May only have 1 indexed object in a prefix list.

      for each field f in p do {
         if f does not already exist in pObject, clone it (as in the
         definition of PInstantiate) and add it to pObject.

         if f does already exist in pObject, then
            PrefixMerge(pObject.f, p.f, pObject.f.residual).
      }.

      residual = ResidualMerge(p.residual, residual).
   }.

   if indexedObject != NIL
      add all indexedObject's indexed fields to pObject.

   Update header fields as appropriate.
   Reduce(residual, pObject).
   pObject.residual = residual.

   return pObject.
}.
```

Figure 4-7. The `PrefixMerge` algorithm.

Specifically, `ResidualMerge` combines expressions in `prefixResidual` and `baseResidual` such that all expressions asserted in either residual are also asserted in the result. Generally, we merge the residuals by conjoining them with the semicolon operator. However, we must handle some special cases, depending on whether the prefix or the base pattern's residuals terminate with an assertion operator (for example, the semicolon) or whether they terminate with an object, thus returning a value.

For residuals that terminate with an assertion operator, we assume that the return value is an implied `self`. There are four cases of interest:

- Neither the prefix nor the base return a value (e.g., they both return `self`);
- The base alone returns a value;
- The prefix alone returns a value;
- Both the prefix and the base return a value.

In the first case, neither the prefix nor the base return a value; this is equivalent to both of them returning `self`:

```
P:   { PExprs; };              or      P:    { PExprs; self };
B: P { BExprs; };              or      B: P  { BExprs; self };
```

In this case, we can concatenate the two sets of expressions to yield the combined object

```
B:  { PExprs; BExprs; };       or      B:    { PExprs; BExprs; self };
```

that simply is equivalent to returning the combined object's `self`.

In the second and third cases, either the base or the prefix returns a value, but not both. In these cases, one of the objects returns an implied `self`, while the other returns a value. When `B` returns a value but not `P`,

```
P:   { PExprs; };              or      P:    { PExprs; self };
B: P { BExprs; BResult };      or      B: P  { BExprs; BResult };
```

or `P` returns a value but not `B`,

```
P:   { PExprs; PResult };      or      P:    { PExprs; PResult };
B: P { BExprs; };              or      B: P  { BExprs; self };
```

the result value is appended to the conjunction of the rest of the expressions, yielding the combined cases

```
B: { PExprs; BExprs; BResult };
B: { PExprs; BExprs; PResult };
```

for `B` or `P` returning a result, respectively.

Since a result can include information about `self` (for example, an expression containing some attribute of `self`), we consider a return result to be more specific than `self`; we thus return it as the object value.

In the final case where both the prefix and the base provide a result, the result must be the same in both cases. We do this to ensure that the combined object's specification produce a single result that satisfies constraints both in the prefix and in the base. The prefix merging process asserts that the two results, `PResult` and `BResult`, are equivalent, and appends either one as the value of the combined expression. Thus merging `B` with `P` in

```
P:     { PExprs; PResult };
B: P   { BExprs; BResult };
```

results in the combined `B`:

```
B: { PExprs; BExprs; PResult = BResult; BResult };
```

Since `PResult` and `BResult` are asserted to be equal, we may return either one as the result of the combined pattern.

4.2 Local Constraint Satisfaction

Siri accomplishes constraint satisfaction by using the same ATR+ augmented term rewriting mechanism it uses for object reduction. Siri provides a set of special constraint patterns that create and modify *ordered linear combinations* [Derman84], an equational normal form. Siri uses these to solve for expression variables that represent object attributes. This is the exact technique that Bertrand uses; Siri's constraint patterns perform the same transformations as Bertrand's rules, and solve constraints in the same way.

4.2.1 Equation Simplification Patterns

Siri performs basic algebra simplification and solving reductions by rewriting complex algebraic expressions. Some constraint patterns simplify expressions using known algebraic identities, while more complicated patterns define objects that match particular equational forms for which a closed form solution is known.

For example, Siri provides the following constraint patterns for simplifying numeric expressions:

```
"0 + b'aNumber":        aNumber { b };        -- Addition of zero
"0 * b'aNumber":        aNumber { 0 };        -- Multiplication by zero
"1 * b'aNumber":        aNumber { b };        -- Multiplication by one
"0 / b'aNumber":        aNumber { b~=0 ; 0 };-- Dividing zero by a number
"a'aNumber ^ 0":        aNumber { 1 };        -- Raising to the zeroth power
"a'aNumber ^ 1":        aNumber { a };        -- Raising to the first power
```

Note that we are not required to return a single numeric result; in the case of dividing zero by a number, we can assert that the number we are dividing by must be zero, and then return the value.

We can also have additional patterns to solve for variables in simple expressions. In particular, we can match on semicolons in labels to determine what is being asserted, and what result is required. To solve a simple linear equation, we can write the following constraint pattern, adapted from [Leler88]:

```
"a'aNumber*x'aNumber + b'aNumber = 0; x": aNumber {
   a ~= 0; self = -b/a; self
};
```

This pattern matches any assertion of the form `a*x+b=0`, where the result required is `x`, and `a`, `b`, and `x` are numbers. To solve this equation we assert that `a` must not be equal to zero, and return the expression `-b/a` as the result.

While the above pattern could be useful in limited circumstances, it is not sufficiently general to handle sets of linear equations, or equations of differing lengths. We need a canonical form for linear expressions so that we can manipulate sets of expressions with an arbitrary number of variables. This canonical form is the ordered linear combination.

4.2.2 Ordered Linear Combinations

Ordered linear combinations are expressions with terms that are maintained in a canonical `coefficient*variable+value` form, sorted left to right lexicographically by the variable names. By keeping expressions ordered by variable, we may manipulate and combine them with other expressions quickly: we can multiply or divide an ordered linear combination by a constant in linear time, or add two together in linear time.

Once we equate an expression to zero we can solve for its variables using simple rules for Gaussian elimination. Augmented term rewriting's binding operation makes it possible to set a variable to a value and thus remove it from the equation. More complex solvers using, say, Groebner bases can be implemented to handle more advanced systems of equations [Winkler85].

Of course, most expressions are not simply linear; however, near-linear expressions are solved as well using the same mechanism. Because of the way that the rules work, Siri cannot match nonlinear terms immediately, but defers matching until the linear subexpressions are reduced, thus making it possible to introduce values to simplify the nonlinear expressions (see section 5.5.1 and [Michaylov92]).

Ordered linear combinations use two special operators to distinguish linear terms from other terms in the expression for matching purposes: variables are multiplied by their coefficients by

**, and are added to other linear terms by ++. These linear terms are matched specially to combine and order them.

For example, constraint patterns transform the expressions

```
F = 9/5 * C + 32; C = F; C
```

into the ordered linear combinations

```
0 = 9/5**C ++ -1**F ++ 32; 0 = 1**C ++ -1**F ++ 0; C
```

by finding the linear terms, rewriting them into terms that use the special operators ** and ++, and then ordering the terms lexicographically by variable name. Once in this normal form, the equation solving patterns can solve for variables using Gaussian elimination steps.

4.2.3 Binding and Solving for Attributes

When attributes do not yet have a value, they are represented by variables in expressions. ATR and ATR+ both provide a *single-assignment* semantics that allows for the one-time binding of variables to values. Binding a variable representing an attribute to a value sets that attribute's value, and removes the variable from any expressions that contain it in the environment.

Binding of a variable occurs in either of two cases:

- when a numeric attribute is equated to a constant value;
- when an ordered linear combination is equated to zero.

The first instance handles the simple case of a numeric equivalence. The constraint patterns that perform this binding is

```
"n'aNumber = k'aConstant ; rest": { n is k ; rest };
"a is b": aPrimitive { %%primType is aBoolean; self };
```

The first pattern matches expressions that equate a numeric object to a constant, and rewrites the expression to a binding expression. The second pattern binds the object a to the value b, and rewrites the expression to true.

For example, the constraint pattern above rewrites the expression

```
a = 10; a
```

to the following:

```
a is 10; a                    -- result of binding pattern
true; 10                      -- a is bound to 10 and replaces occurrences
10                            -- true is rewritten away
```

Once Siri binds a variable representing an attribute to a value, it replaces every occurrence of the variable with that value. This step removes the variable from all of the expressions in which it participates, allowing further expression reduction.

The second case is the general Gaussian elimination step for ordered linear combinations. The constraint pattern that performs this step is

```
"0 = ((c**v'aNumber) ++ rest'aNumber) ; moreExprs": {
      (v is ((-1/c) * rest)) ; moreExprs };
```

which simply isolates the matched variable by performing the appropriate algebraic steps.

Note that our simple example expression

```
a = 10; a
```

can be rewritten to the ordered linear combination

```
0 = 1**a ++ -10; a
```

and then solved with the more general ordered linear combination solver. However, for efficiency, Siri's equation solver includes the numeric equivalence pattern to perform the simple numeric bindings in a single step. The general Gaussian elimination pattern would require multiple rewriting steps to transform the expression into the normal form, and eventually to bind the variable.

To illustrate binding and equation solving, the following outlines the rewriting steps taken to solve for `C` in our previous example. In the expressions

```
F = 9/5 * C + 32; C = F; C
```

we are solving for `C` (and `F`), where `C` and `F` have the same value, and are related with the standard Celsius/Fahrenheit conversion. During the reduction process, `C` and `F` will be bound to values as part of the constraint satisfaction process; by appending `C` to the end of the expression, it will be rewritten to its value and returned.

The constraint patterns rewrite the first of the given expressions into an ordered linear combination, leaving the second two expressions unchanged:

```
0 = 9/5**C ++ -1**F ++ 32 ; C = F; C
```

The Gaussian elimination pattern matches, rewriting the expressions to

```
0 = 9/5**C ++ -1**F ++ 32 ; C = F; C
C is -1/(9/5) * (-1**F ++ 32); C = F; C
```

which Siri reduces further to

```
C is (5/9**F ++ -160/9); C = F; C
```

Now, Siri binds `C` to the expression `5/9**F ++ -160/9`, replacing all occurrences of `C`:

```
5/9**F ++ -160/9 = F; 5/9**F ++ -160/9
```

Constraint patterns order the leftmost expression and set up a binding for `F`, resulting in

```
0 = -4/9**F ++ -160/9; 5/9**F ++ -160/9
F is -1/(-4/9) * -160/9; 5/9**F ++ -160/9
F is -40; 5/9**F ++ -160/9
```

`F` binds to the value `-40`, and rewriting continues to

```
5/9**(-40) ++ -160/9
-40
```

which is the value at which degrees `C` is equal to degrees `F`.

When variables are bound to values, the rewriter replaces instances of the variable with its value not only in the current expression being rewritten, but also in all residuals in all objects that reference the variable as well. For example, in

```
easyMath2: {
   a: aNumber { self = z - 10; self };
   b: aNumber { self = a + 14; self };
   c: aNumber { self = z + 4; self };
   z: aNumber;

   a + b = 2*z + c;
};
```

Siri may initially bind `a` to `z - 10`. Doing this rewrites all instances of `a` in all residuals to the expression `z - 10`: in the subobject `b`'s residual (the term `a + 14`); in `a`'s own residual (the term `self = z - 10`, where `self` represents `a`); and in `easyMath2`'s residual (the term `a + b = 2*z + c`).

For more details on the equation solving patterns and ordered linear combinations, see Appendix B for the Siri code that implements them.

4.2.4 Redundant and Conflicting Expressions

Redundant expressions are those that describe the same relationship in a different way. Conflicting expressions describe relationships that cannot all be satisfied at the same time. Rather than being problematic, redundant and conflicting expressions can be quite useful in the constraint satisfaction process.

Redundant Expressions

Redundant expressions in constraint patterns can be used for several purposes. First, the programmer may introduce a redundant expression to give the solver a direct solution to an inverse function that the solver could not derive itself. For example,

```
y = sin x; x = arcsin y;
```

are redundant expressions, and if the solver cannot solve trigonometric equations, it can use the above expressions to solve for one variable in terms of the other. Note that Siri does not need to know that these two are inverses of each other; the appropriate expression is simply used if required.

The equation solving mechanism writes away redundant expressions if it can and only maintains the information that is needed in order to solve for the desired variables. For example, if we bind a in the redundant expressions

```
a = 2*b; b = a/2;
```

we observe the following reductions

```
true; b = (2*b)/2;        -- having bound a
b = b;                    -- having reduced b = (2*b)/2
true;                     -- having reduced b = b
(empty expression)        -- after asserting true
```

Thus, redundant expressions are harmless in the worst case, and helpful if used to provide inverses for otherwise unsolvable expressions.

Conflicting Expressions

If, during the equation solving process, two or more equations are in conflict, Siri will raise an exception (basically, an equation which is `false` will be asserted to be `true`). Such a conflict indicates a semantic error, requiring programmer intervention to solve.

The discovery of conflicting expressions is useful in the domain of multiple inheritance. If a constraint pattern inherits an attribute `A` from several prefixes, the `PrefixMerge` process (see section 4.1.4) assumes that the attributes from each prefix should be compatible with each other. If in the process of evaluation a conflict occurs, Siri notifies the programmer that there is a problem inheriting the "same" attribute from two different sources, and the attribute must either be renamed, or the reason for the inequality must be solved in the prefixes.

For example, we might have two constraint patterns

```
P: {
   x: aNumber;
   A: aNumber { x - 16 };
   ...
};

B: P {
   A: aNumber { x + 4 };
   ...
};
```

The combined attribute `A`, after `PrefixMerge`, would have two equations in its combined body (remembering the implied `self` references):

```
A: aNumber { self = x-16; self = x+4; self }
```

This clearly leads to a contradiction (the assertion that `4 = -16`) and thus the inheritance is not compatible.

Rewriting Recursive Patterns

One issue in constraint satisfaction using augmented term rewriting comes up when a pattern is recursive or mutually-recursive. The object may not be able to be completely reduced in this case due to the recursion, since the recursion may be infinite. To avoid this, there are several solutions: the first is to prohibit completely any recursive rewrites on an object at reduction time, which requires simply that the same object never be matched in its own reduction; the second is to limit the number of times an object is rewritten in the pre-instantiation reduction process; and the third is to keep track of all objects used during a reduction, and avoid matching objects that have already been used by maintaining an active set of reduction objects,

and removing objects used previously in the process. In the prototype version of Siri, we use the first solution of prohibiting any recursive rewrites at reduction time.

The binding process makes it possible to handle mutually-recursive patterns. Say we have two objects, x and y, defined simultaneously:

```
x: aNumber { 27 - y };
y: aNumber { x - 16 };
```

In standard term rewriting, if want to solve for x, we rewrite x to its body, followed by y, we would observe the rewriting sequence

```
x
27 - y
27 - (x - 16)
27 - ((27 - y) - 16)
...
```

which never terminates.

However, recall that in ATR+ the given patterns are equivalent to the patterns

```
x: aNumber { self = 27 - y; self };
y: aNumber { self = x - 16; self };
```

which include the implied `self` references. These `self` references make it possible to solve these mutually-recursive equations directly. We can bind x to its value `27-y`, and replace all occurrences of x with its value. When its value replaces the variable in y's residual, we replace the occurrence of y that appears in x's residual with `self`. Then we have

```
x: aNumber { ...bound to 27 - y... };
y: aNumber { self = (27-self) - 16; self };
```

Now we can easily solve for y, and back-substitute y's value into x's residual to solve for the value of `x (21.5)` and `y (5.5)`.

4.3 Modifying Objects by Method Evaluation

Objects reevaluate their internal state in response to client messages. This process of reevaluation is similar to other constraint satisfaction mechanisms such as propagation of values. However, Siri recomputes the state of an object by solving a combination of permanent object-based constraints and temporary constraints that hold during method evaluation. This recomputation then rebinds the object's attribute values, thus changing the object's state.

Methods defined in an object *fix* attributes and *assign* values to attributes. Fixing an attribute sets up a constraint that equates the attribute to its current value, thus preventing it from changing. Assigning a value to an attribute deletes its current value, making the attribute free, and then sets up a constraint equating the attribute to its new value.

The set of constraints to be evaluated come from three sources: constraints that fix and assign values in the method; constraints that hold during evaluation; and constraints that are defined in outer scopes. Siri solves these constraints to evaluate the method, and to rebind the attributes to the appropriate values.

4.3.1 Method Reduction

Siri handles the reduction of constraint expressions in method objects differently than in other kinds of objects. Any binding of variables to values is postponed until the method is evaluated. In the example of `aRectangle`, we invoke the `initially` method when we create an instance of `aRectangle`:

```
aRectangle {
   ...
   initially: aMethod {
      width = 100; height = 50; center = point (400, 300);
   };
   ...
};
```

The residual of the method `initially`, after reduction, is

```
right=450; left=350; top=275; bottom=325;
```

The binding rule for methods prohibits the attributes from being bound at reduction time. If we fully reduced the expressions, each of the equality assertions would result in an attribute binding. By postponing binding until we actually evaluate the method, we can use the constraints within the body temporarily, binding values for attributes only at that time.

Note that Siri reduces the `initially` method in this case without reference to any outside values; thus, Siri can reduce it at compile time to the final equations given above. While the programmer could have written this directly, the intent of the programmer is clearer in the constraint-based code.

4.3.2 Method Invocation

At method invocation time, for each fixed attribute, Siri substitutes an equation asserting the attribute to its current value, and rewrites the attributes themselves to their residuals. Siri then evaluates the resultant expressions, binding attributes to new values. The method retains the

residual expression for reevaluation at the next invocation time. For example, Siri rewrites the residual for the method moveTo in aRectangle (from section 1.5.1),

```
"moveTo newCenter'aPoint": aMethod {
   width, height fixed;
   center = newCenter;
};
```

to the following, using the constraint pattern that defines point equality and the definition for center:

```
width fixed; height fixed;
center.x = newCenter.x; center.y = newCenter.y;
```

At evaluation time, Siri substitutes the newCenter into the residual and sets up the fixed constraints. If we assume that width is 100, height is 50, and newCenter is (180, 230), the subject expression for evaluation becomes:

```
width = 100; height = 50;
center.x = 180; center.y = 230;
```

Siri then substitutes the residuals for width and height (shown in bold below),

```
right-left = 100; bottom-top = 50;
center.x = 180; center.y = 230;
```

and the bindings for the two center coordinates, which had been reduced from their original definitions to simpler ones involving only the right and left attributes (again, shown in bold):

```
right-left = 100; bottom-top = 50;
(right+left)/2 = 180; (bottom+top)/2 = 230;
```

Now, this expression only contains the four attributes top, left, bottom, and right. Siri then solves for them and binds their new values using Gaussian elimination, as explained in section 4.2.3, and the method invocation is complete.

4.3.3 Value Dependencies

The changed message is Siri's dependency mechanism for notifying dependents that a shared object has changed state. When this occurs, Siri checks to see if the object has any dependents that are pointing to it via a shared attribute. If there are dependent objects, Siri automatically invokes the appropriate changed method on those objects, giving them the chance to update their own state if they so desire.

Siri maintains dependencies in a single system-wide table, mapping each object to a list of its dependents. If the list of dependents is longer than a few entries, Siri performs a heap walk instead of storing the dependents directly. The walk is done by scanning the heap for references to the shared object, and then scanning backward to the beginning to find its object identifier. SELF uses this heap-scanning scheme [Chambers89] for finding objects that need to be recompiled due to changes in inlined code definitions.

4.3.4 Structure Dependencies

In cases where an object's structure or code changes, rather than simply its state, Siri must notify and reevaluate an additional set of dependents: those that use the object's definition as a prefix or a nested part, as well as the object itself. This is an extension of the value dependency problem addressed in section 4.3.3, and is handled in a similar way.

4.4 The Object Structure

The lowest level structures in the Siri environment are the memory structures of objects. These structures represent the reduced value of constraint patterns.

The in-memory structure of a Siri object is patterned after the tagged object structure in languages such as Oaklisp [Lang86], where an object consists of a set of consecutive *fields* in memory that store the object's data. Each field is a 32-bit *object identifier* that may either represent an *immediate* data object or an address of a *boxed* object elsewhere in memory.

4.4.1 Object Identifiers

The low order two bits (bits 0 and 1) in an object identifier are the object's *tag*. This tag specifies the meaning of the remaining 30 bits.

If the tag value is zero, then the identifier encodes an immediate object; bits 2 and 3 are then a *subtag* that specifies the meaning of the remaining 28 data bits. There are currently only two variants of immediate objects: a 28-bit immediate `integer`, and a 3-byte packed `string` object.

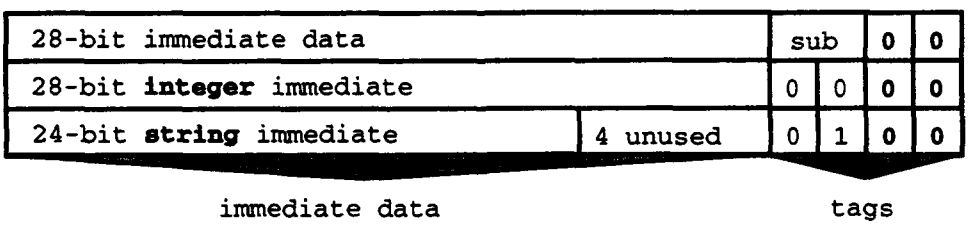


Figure 4-8. Immediate objects.

If the tag value is not zero, then the high order 30 bits of the identifier is a pointer to a boxed object elsewhere in memory, aligned to 32-bit words (e.g., the low two bits of the address are masked to zero).

The three variants of boxed objects are *shared* objects, *copy-on-write* objects, and *owned* objects (Figure 4-9). Identifiers with a *shared* tag point to objects that are considered to be common state. Any modifications to the referenced object are visible to all referrers. Identifiers with a *copy-on-write* tag point to privately-owned objects that are being referenced temporarily for the sake of efficiency. If such an object is to be modified, then it is cloned before the modification, with the new object replacing the referenced one. Finally, identifiers with an *owned* tag are objects accessed exclusively by the enclosing object. They define private information that is known not to be referenced outside of the scope of the object, and is considered to be part of the object's state.

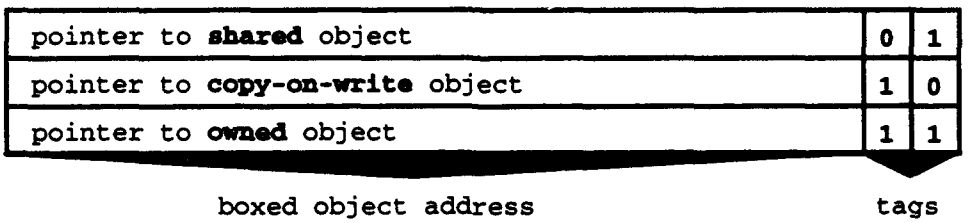


Figure 4-9. Boxed objects.

4.4.2 Basic and Extended Object Headers

Every object has a header, or fixed set of fields, that Siri maintains. There are two variants of an object header: a basic header and an extended header (Figure 4-10). The basic header contains information for evaluation of intrinsic, system-defined objects. The extended header contains additional information for variable-length and user-defined objects.

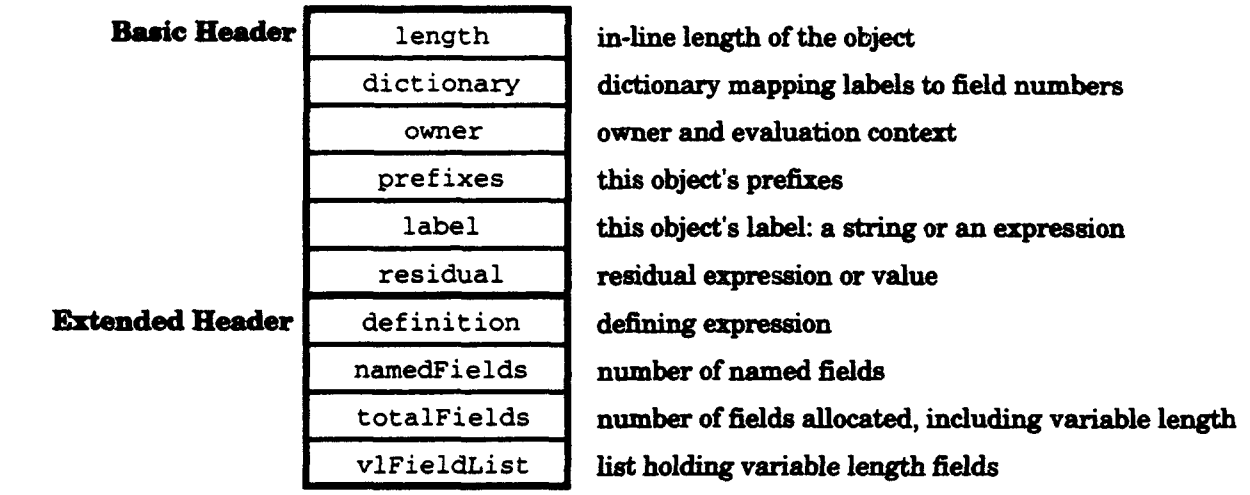


Figure 4-10. Basic and extended object headers.

4.4.3 Objects with Basic Headers

Objects with basic headers have a fixed number of fields, do not have a user-viewable definition, and cannot be extended at the user level. Basic objects are fixed length so that the Siri interpreter can use predefined offsets in the objects to find fields without a name lookup. Many of the Siri system's intrinsic objects—`aHashTable`, `aList`, `anOperator`, `aStack`, `aScanner`, `aParser`, and `aPrimitive`—are basic objects (see Appendix B). There are six fields in a basic object header: the `length`, the `dictionary`, the `owner`, the `prefixes`, the `label`, and the `residual`.

The `length` field is an immediate `integer` object that specifies the number of contiguous object identifiers in memory. Note that the length of an object is not the number of user-defined fields in the object, since the first six to ten fields in an object are used internally, and there may be additional variable-length fields. The `length` field is marked with a special immediate tag that indicates the start of the object; Siri provides this special treatment for `length` so that the beginnings of objects can be found in a heap scan (see section 4.3.3). For the purposes of this discussion, however, we will assume that the `length` field is a simple `integer` immediate.

The `dictionary` field points to a copy-on-write or owned `dictionary` object that maintains a mapping between field names and field indices within the object. A group of objects may use a single dictionary as long as their layouts remain unchanged, with no fields added or removed. An object may be found by looking up its name in the dictionary to find its field index, and then finding the object identifier at the given offset.

The `owner` field points to the object that encloses this object lexically. The owner object defines the object's evaluation environment. When looking up the value of a name, the dictionary of object itself is checked, followed by the lexically enclosing objects defined by the owner, the owner's owner, and so on.

The `prefixes` field is an owned list object that lists the object's prefixes. This field is used for name lookup as well; names are looked up in the chain of prefixes first, and if it is not found, the lookup proceeds to the lexically-enclosing objects in turn.

The `label` field is an owned `string` or `expression` object that indicates the object's label. If the label is a string, then the object is simply named by the string, and may be looked up in the dictionary directly. If the label is an expression, then the dictionary stores the label expression as a string, just as is done with a simple label; in addition, the expression is parsed and stored as an expression object in a structure maintained by the system for matching and reduction purposes.

The `residual` field is an owned `expression` object that stores the residual of the object's evaluation. Often the value of an object's residual is `self`, which represents the object in its entirety.

For an example of a basic object, consider a list object `myList` that is a part of another object called `test1`. The `myList` object is an instance of `aList` that defines a `head` and a `tail`:

```
test1: {
   myList: aList;                    -- aList is like a LISP cons pair
   ...
};
```

length of object = 8 fields	0	0	0	0
dictionary = (dictionary defined by aList)			1	0
owner = test1			1	0
prefixes = aList			1	1
label = myList			1	1
residual value = self			1	1
head of list			1	1
tail of list			1	1

Figure 4-11. A list object stored in `test1`.

Figure 4-11 shows how test1 is stored in memory. An instance of `aList` consists of a basic object header, and adds fields to store the `head` and `tail` of the list as indicated. The `dictionary` and `owner` objects are shared among other instances of `aList` and other objects in the same scope respectively. All other fields are owned by the object itself.

4.4.4 Objects with Extended Headers

An object with an extended header adds two features to a basic object: it stores its user-level definition, and it can have a variable number of fields. Objects in Siri with extended headers include arrays, strings, objects that act as evaluation environments for other objects, and user-defined objects.

The extended header adds four new fields to the basic header: the `definition`, the `namedFields`, the `totalFields`, and the `vlFieldList`.

The `definition` field contains the object's original body before it is evaluated. For example, the `definition` field for the object `aKCFTemp`

```
aKCFTemp: aCFTemp {
   K: aNumber;
   K = C + 273;
};
```

is simply the expression

```
K: aNumber; K = C + 273;
```

The next three fields, `namedFields`, `totalFields`, and `vlFieldList`, provide variable-length information. There are two different kinds of object fields that can vary in number: the object may have an arbitrary number of *named* fields, and it may also have an arbitrary number of *indexed* fields. For example, a string object has a set of fixed named fields, like other objects, including a `length`, a `dictionary`, and so on; however, in addition it has a variable number of indexed fields that store the string's bytes. The `namedFields` field specifies the number of named fields, while the `totalFields` specifies the total field count, named as well as indexed. In addition, if the `totalFields` of an object is greater than the object's `length`, then there are (`totalFields - length`) fields that are stored in a list pointed to by `vlFieldList`. Fields whose index is less than `length` are looked up directly, by indexing into the object. Fields whose index is greater than the length are looked up in the linked list pointed to by `vlFieldList`. This allows the object to increase size without having to move to a new location in memory, as would be required if the object were to be reallocated.

An object with seven indexed fields, four of which are stored in the object and three stored in the `vlFieldList`, would appear in memory as follows:

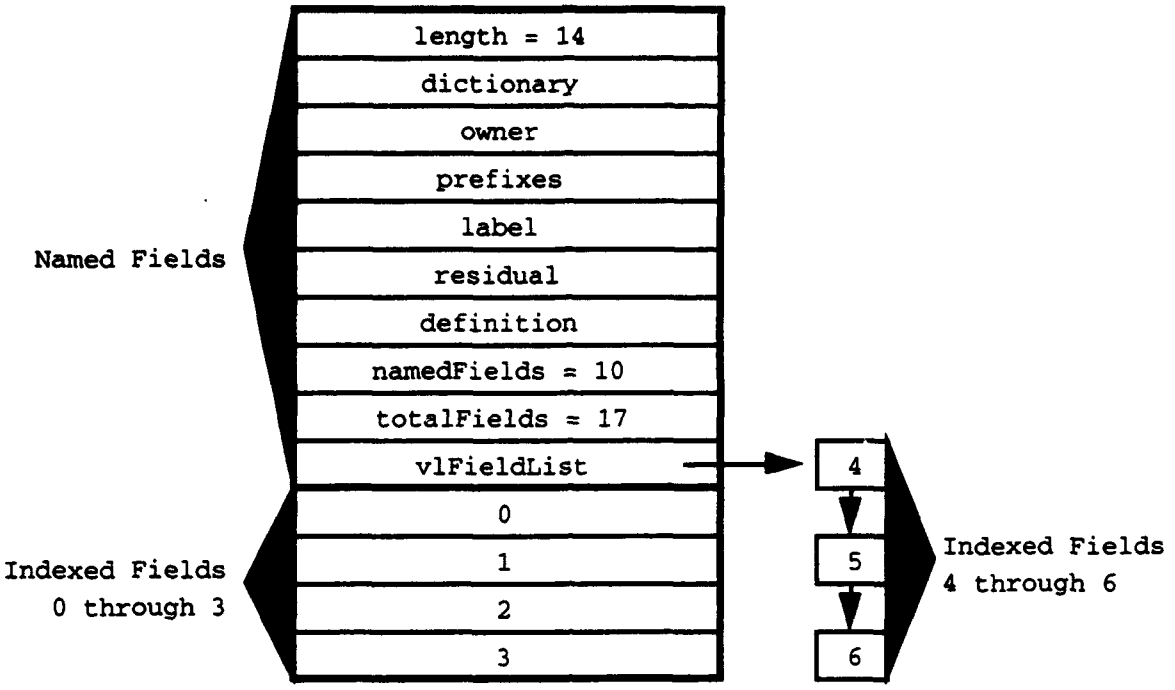


Figure 4-12. An object with a `vlFieldList` holding three fields.

In this object, the first indexed field is at an offset of 10 in the object's memory block. `vlFieldList` stores fields with indexes greater than three.

An example of an object with an extended header is the string `myString` with the value `"Siri"` stored in an object called `test2`:

```
test2: {
   myString: "Siri";
   ...
};
```

`myString` would be stored as follows in memory:

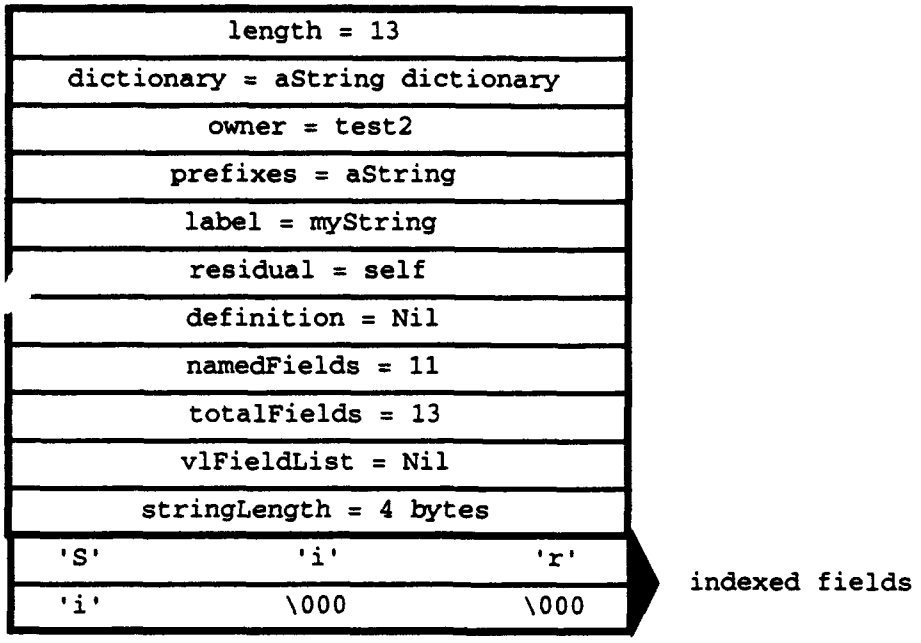


Figure 4-13. The string "Siri" stored in test2.

Following the basic header, the extended header adds four fields, and the string object itself adds three fields: one named field that encodes the length of the string, and the string data stored in two consecutive indexed fields.

Because strings are variable length, we can modify this string by appending new characters to the end; if the size becomes greater than 6 bytes, then an additional field is created to store the new characters by adding an entry to the `vlFieldList` list and increasing `totalFields` by 1.

An alternative implementation for strings can be used as well that would not require the string length to be maintained. Since `totalFields - namedFields = 2`, we know that there are two indexed fields in this string, so the length of the string is 4, 5, or 6 bytes (storing three bytes per field). By null-terminating the string we can quickly determine the exact length (including null byte) without scanning the entire string, by indexing to the last three characters and then checking each of the three possibilities.

Having 11 fields of overhead per string is quite wasteful, and access to fields stored in `vlFieldList` can be quite slow. However, in chapter 5 we will describe how much of the common header information can be factored out, and access time to fields can be improved.

4.5 Implementation Summary

In this chapter, we have seen how Siri's extension to Bertrand's augmented term rewriting engine reduces and evaluates constraint patterns. We described how Siri performs constraint satisfaction by rewriting expressions to a normal form, and then performing Gaussian elimination on variables in the expressions. We outlined how Siri's methods provide temporary constraints that direct the reevaluation of objects to new states. Finally, we discussed the in-memory organization of Siri objects.

Chapter 5
Optimizations and Improvements for Siri

This chapter discusses some potential optimizations to improve the performance of constraint pattern evaluation. In a partially-declarative language like Siri, programs can be expressed in abstract ways, leaving it up to the system to find a way to execute them efficiently. Just as object-oriented programming demands machine resources by requiring the system to make the connections between objects and their corresponding methods, constraint pattern programming is even more demanding, since it requires the system to create, and then evaluate, a concrete implementation given an abstract specification. In particular, Siri must perform algebra to solve for object values.

To make Siri acceptable for experimental purposes, Siri must be fast enough, capable enough to solve the kinds of problems that one would like to use Siri to solve, and compact enough for use on standard workstations and personal computers. Of course, *fast*, *capable*, and *compact* are relative terms; we would like to aim for a system that can be used for small to medium-sized, non-computationally-bound projects for experimenting with Siri's programming concepts on a personal computer.

The optimizations suggested in this chapter are not novel; they are either straightforward or borrowed from published techniques.

5.1 Current Performance and Capability Limitations

The current implementation of Siri is unoptimized, and as such is limited in its speed and capability. There are four major areas in which this system is limited:

- Memory-intensive structures;
- Slow evaluation engine;
- Redundant code execution; and
- Limitations in the constraint solver.

These problems limit Siri's usefulness; the following subsections will describe these limitations more fully, and outline why the current implementation has them.

5.1.1 Memory-Intensive Structures

Every object in a Siri system currently has a header that is six to ten words long describing the basic attributes of the object: its length, information about its fields, and pointers to expressions and other objects that define it. Even small objects require a lot of space; for example, a single element in a linked list currently requires 8 32-bit words of storage, four times as much storage as the two pointers required for a cons pair's CAR and CDR. This is quite wasteful, and unacceptable for programs that handle large amounts of data. Since we would like to run Siri programs on modest hardware, it is desirable to make Siri objects, and the resulting programs, more compact, both so that they will fit in limited memory and to improve garbage collection performance.

Section 5.2 will examine how we can factor out common object data to minimize the size of individual objects, without incurring too great a performance penalty in accessing the shared information.

5.1.2 Slow Evaluation

The basic evaluation mechanism in the current Siri implementation is the ATR+ term rewriting engine. Rewriting is used both for reduction and evaluation of objects; the ATR+ engine in turn consists of a pattern matcher, object instantiator, and subtree rewriting subsystems. Rewriting in Siri is quite slow, and there are several reasons for this. The first reason is that the pattern matching system is a brute-force tree/subtree matcher that recursively tries to match all possible labels that are visible in the current environment. Finding these labels, as well as simple names, is done through a lookup process that scans dictionaries of all objects that define names in the given scope. Neither the pattern matcher nor the name lookup mechanisms have been optimized at all, contributing to the glacial performance of the rewriter. In addition, rewriting by itself is quite inefficient compared to evaluation of compiled machine code, even if the pattern matcher, instantiator, and subtree rewriters were highly optimized. In order to speed up evaluation, we need to be able to replace rewriting with faster evaluation mechanisms where possible, and to speed up ATR+ itself in the cases that it is still used.

In section 5.3 we examine how better algorithms, including compilation of satisfaction methods, can be used to speed up the evaluation process as a whole, and as a tradeoff with their additional space requirements.

5.1.3 Redundant Code Reevaluation

Aside from pure execution speed concerns, there are issues involving the use of Siri programs in user and programming environments, where reevaluation of constraint patterns after modification often results in redundant code execution. In such environments, often objects are edited and changed in small increments; though these structures are large and interconnected, only small pieces of the data are modified during interaction with the user, and large parts of the computation will have been done previously. Consider editing a large document in a word

processing program: typing text into a window usually changes only a small part of the system's model of the document (typically, adding characters to a line), requiring only small amounts of computation. In some cases, however, the system will be required to reevaluate a much larger part of the model (for example, causing a word, line, or paragraph break; or changing a paragraph style's formatting, which requires the entire document to be laid out from scratch).

Also, an object may need to be reevaluated when its structure changes, such as when a new subobject is added or when a constraint is changed. Because these changes do not change only values, the entire object as well as any structural dependents, such as objects that use it as a prefix, must be recomputed.

For Siri to perform well in such environments, we would like a general mechanism for minimizing computation during reevaluation of large models that takes advantage of previously-evaluated sections.

Section 5.4 discusses *incremental computation* which can be used to improve Siri's performance in such situations.

5.1.4 Limited Constraint Solver

The constraint solver provided in the current Siri system is a simple linear solver based on ordered linear combinations, written in Siri itself. This solver can handle linear and slightly non-linear expressions, can solve for variable values, and can handle multiple simultaneous equations. Unfortunately, the current solver cannot handle inequalities or nonlinear equations, both common occurrences in applications such as graphical user interfaces or scientific programming. In addition, the solver handles only a fixed set of constraints; there is no hierarchy of constraints with different weights.

Section 5.5 considers ways that the constraint solver can be strengthened, and how a variety of solvers can be plugged into the Siri system to perform special-purpose algorithms for constraint satisfaction in well-known cases.

5.2 Space Optimizations

To minimize the space required for a given Siri program, there are three major approaches that can be taken: factoring out common header fields; sharing of special objects such as dictionaries and expressions; and trading speed for space by computing attribute values rather than storing them in object fields.

5.2.1 Minimizing Required Header Information

Every Siri object currently has six to ten header fields as described in Chapter 4. The fields in basic, system-defined objects are:

- `length`, the length of the object in memory;
- `dictionary`, a pointer to the dictionary mapping field names to field offsets;
- `owner`, the object defining this object's environment;
- `prefixes`, the object's prefixes;
- `label`, the object's name (either parameterized or symbolic);
- `residual`, the object's residual value

Extended and variable-length objects include four more fields:

- `definition`, the object's original body expressions;
- `namedFields`, the number of named fields in the object;
- `totalFields`, the total number of fields, named and indexed, in the object; and
- `vlFieldList`, a list holding the variable length fields not contiguously allocated.

One way to minimize the number of required header fields in each object is to factor out shared information into *map* objects [Chambers89] or *families*. A family object holds information that is common to a group of objects; thus a single pointer in each object can point to that object's family, which stores the shared information.

Fields that are candidates for factoring out into families include the `prefixes`, the `dictionary`, the `definition`, and the `namedFields` (unused for objects with basic headers). We can replace these four fields by a single pointer to a `family` object.

It is easy to detect syntactically when an object being created can become a member of an existing family. Any object that is defined by a constraint pattern of the form

```
newObject: anExistingObject;
```

is a member of the family defined by `anExistingObject`, since it is a simple clone, with the exact same fields and expressions. Objects defined by constraint patterns of the form

```
newObject: aPrefix { aBody };
```

will instead define a new `family` object, since `aBody` may include new fields to be added to the initial definition provided by `aPrefix`, and may also provide new expressions. Instantiations

of `newObject` then point to the newly-defined family, as above. In the case of multiple prefixes, the family simply maintains a list of prefixes as well; we can easily locate their respective families by looking up the `family` field in each prefix object.

Another way to slim down the required header is to remove information that can be found in other places in the system. For example, an object's dictionary stores all of the subobject labels as key entries; by organizing the dictionary such that a lookup on either the key or the offset is possible, we can retrieve an object's `label` by finding its offset in its owner, and then finding the key that corresponds to its offset value in the dictionary. Similarly, the `totalFields` count is not necessary, since we can compute the total number of fields by adding the object's length to the length of the list pointed to by `vlFieldList`. Finally, we can omit the `owner` field as well by rewriting the evaluation engine to keep track of owners as subobjects are evaluated. The current Siri system includes these fields for ease of implementation, as well as for intentional redundancy for consistency checking of structures.

By factoring out family information, and deleting redundant fields, we reduce the basic header for intrinsic objects to the following:

- `length`, the length of the object in memory;
- `family`, a pointer to the object's family; and
- `residual`, the object's residual value.

which is a savings of 3 fields out of the original 6 in the basic header. Variable length objects simply include one additional field, `vlFieldList`, and use the `namedFields` entry in the `family` to determine where the variable length fields begin. This represents a savings of six fields out of the original ten in the extended header. In addition, variable length objects that store fields in the `vlFieldList` can reduce their overhead by inlining the external fields at garbage collection time. This improves both compactness and time access to fields.

Reducing the number of header fields does require some additional implementation complexity. The system must create and maintain the family objects, adding and changing them as appropriate; for example, if an object has a field added or removed at runtime, then the system must create a family that reflects the object's new organization. We need to optimize the dictionary implementation to allow lookups on the value as well as the key of a key/value pair to allow the deletion of the `label` field. Finally, we have to restructure the evaluation engine itself so that the current `owner`, or reduction environment, is maintained as part of the evaluation state.

5.2.2 Sharing Expressions

We can also save space by sharing other objects that are not part of the family information. In particular, we can share expressions and subexpressions among related objects.

For example, derived objects include the expressions defined in their prefixes. Rather than copy the expressions down to the derived object in combining them with its body, we could leave a pointer to the prefix expressions in the combined expression. Given the definitions for `aCFTemp` and a derived object `aKCFTemp`,

```
aCFTemp: {
   C, F: aNumber;
   F - 32 = 9/5 * C;
};

aKCFTemp: aCFTemp {
   K: aNumber;
   K = C + 273;
};
```

the residual for `aKCFTemp` is the expression

```
F - 32 = 9/5 * C; K = C + 273;
```

since, during reduction, the system creates the objects `C`, `F`, and `K` and rewrites away their defining subexpressions from the residual. Then, instead of copying the expression from `aCFTemp` to combine it with the expression in `aKCFTemp`, we can store

```
<pointer to aCFTemp residual>; K = C+273;
```

which would represent the same combined expression.

In combination with other optimizations, this one is easy to implement: we can use lazy structure sharing (see section 5.4.2), if provided in the low-level system, to detect instances where this could be used with little or no additional complexity. In addition, sharing subexpressions can also provide significant performance benefits in some situations: see section 5.3.1 for a discussion of Jungle Evaluation [Hoffman88], a technique for building and rewriting shared expression graphs.

5.2.3 Computing Attribute Values

A satisfaction method typically evaluates an object's attribute by computing it as a function of one or more of the object's other attributes, and then storing its value in an object field. In some cases, this satisfaction method can be shared among all members of a family.

For example, in the rectangle object, we can store the `top`, `left`, `bottom`, and `right` attributes in each object and use satisfaction methods to compute `width`, `height`, `topLeft`, and `bottomRight`. These methods can be stored in the object's `family`, just as Smalltalk methods are typically stored in the object's class [Goldberg83]. By doing this, we can compute the

attributes `width`, `height`, `topLeft`, and `bottomRight` on demand and not store their values at all, thus saving four fields in each `Rectangle` object. This is a time-space tradeoff that should be carefully examined and instrumented, since the opposite strategy, caching of attribute values, might be preferable in some situations.

5.3 Speed Optimizations

In this section, we describe several categories of approaches for improving the performance of a Siri system. The first includes techniques for improving the speed of the ATR+ engine itself; the second includes ways to generate code for more efficient instantiation of objects; and the third outlines mechanisms for providing hints to the evaluation engine to specify directly the roles that certain objects play.

5.3.1 Improving Augmented Term Rewriting in ATR+

There has been some recent work on optimizing the process of tree matching and term rewriting [Kosaraju89, Cai90, Ramesh92]. Some of these techniques can work well together by sharing features required in more than one technique; for example, incremental computation uses structure sharing, which can also be used in jungle evaluation.

In Siri, initially we can take advantage of the simplest techniques that provide the best payoff. These techniques include faster pattern matching, techniques for threaded evaluation, a restricted syntax, sharing of namespaces, and a special evaluation technique that reduces redundant rewriting.

Fast Pattern Matching

One of the easiest and most profitable ideas to take advantage of is Bertrand's fast pattern matching mechanism [Leler88]. Instead of performing a general tree matching process, we can take advantage of the restriction that we made earlier on labels, strict left sequentiality, to simplify tree matching to a linear string matching algorithm, using a fast string matcher such as Aho-Corasick [Aho75].

We first flatten labels and expressions into a string by traversing them in preorder, creating nodes that consist of an operator or type, and an annotation that specifies how to find the next node in the tree. Next, we compile all constraint pattern labels into tables, organized by pattern namespace contours. These tables define instructions for an automaton that performs a left-to-right string matching process in the flattened expression, comparing nodes in the expression with nodes in the table. Once a match is found, the instantiator replaces the matched expression, as before.

By compiling the labels into a table for a finite state automaton, we can perform matches very quickly: the time for a match is proportional to the length of the expression being matched, and is independent of the number of labels that might potentially match. Because of the way

constraint patterns are organized, the expressions being matched are usually quite short. Thus, we expect that a matching process based on this idea might be fast enough in general so that other limitations of the evaluation mechanism would dominate.

Merging Dictionaries

When looking up a name in a given scope, the current Siri implementation performs a brute-force search looking up the name in object dictionaries, starting at the current object and proceeding through outer objects, both prefixes and owners. Since there are a small number of entries in each dictionary, finding a name's value may require searching a large number of different dictionaries.

Instead, we can merge object dictionaries into a single large naming structure that overlaps object scopes. We perform a single lookup on a name, returning a second dictionary mapping scopes and values for that name. A second lookup given the current scope returns the object's value in that scope. This technique reduces an arbitrary number of lookups per name to two.

Jungle Evaluation

Jungle evaluation [Hoffman88] is a technique that merges expressions with shared subparts into a single graph. If the term rewriter reduces a subexpression shared among several objects, the result is available to all of the objects that share it. By rewriting subexpressions in such a graph, the performance of the rewriting system as a whole can be substantially increased. For example, the rewriting of the expressions in `aCFTemp` to an ordered linear combination (section 5.2.2) need only be done once if they are shared with `aKCFTemp`. However, if they had been copied, this work would have to be done twice: once in `aCFTemp`, and once in `aKCFTemp`.

Thus, jungle evaluation compresses several term rewriting steps (the rewriting of the expression in each object) into one (rewriting only the shared expression), thus avoiding redundant computation. For this to work in Siri, however, the constraint patterns that are applied to reduce a subexpression must be the same constraint patterns that would be used in each of the different contexts. This is not necessarily guaranteed; for example, the same subexpression in different objects might be rewritten in significantly different ways depending on the constraint patterns visible in their scopes. For example, an object sharing a subexpression in a prefix might define its own constraint patterns that match the subexpression and reduce it to a different residual that is irreducible in the prefix, but reducible in the derived object.

Jungle evaluation has the potential to improve performance significantly in a system like Siri, since Siri's inheritance structure causes objects to share significant parts of their definitions. Jungle evaluation also works together well with other techniques such as incremental recomputation (section 5.4). Jungle evaluation is a generalization of an idea in [Hoffman85], whose equational interpreter hashed instances of new expressions and coalesced new identical terms with existing identical terms, thus sharing in a way similar to hashed consing [Pugh88].

Other Optimization Techniques

The current Siri implementation allows user-definable operators with arbitrarily complex syntax. Because an expression can have any form, we have to perform pattern matching in the reduction process. If we were to restrict the syntax of expressions to a given form, we might be able to limit the pattern matching to the parts of the program that have algebraic expressions, and perform standard compilation processes and name lookups on user-defined constraint patterns. For ideas on how one might do this, see [Chambers92] for a structured, object-based mechanism for handling multimethods.

5.3.2 Pattern Customization

We can use pattern customization, closely related to SELF's method customization [Chambers90], to optimize constraint pattern code based on the code's parameter types. By making multiple copies of a constraint pattern, each with a different but downward-compatible type signature, we can compile specialized versions to take advantage of the specific types in the parameter list. For example, the constraint pattern

```
"a'aNumber someOp b'aNumber":    { ... };
```

can be customized by the system to each of the following special patterns as well:

```
"a'aNumber someOp b'anInteger":  { ... };
"a'anInteger someOp b'aFloat":   { ... };
"a'aFloat someOp b'aNumber":     { ... };
```

as well as many others, where the types of the parameters are all subtypes of `aNumber`. Siri can then use the type annotations of the variables `a` and `b` to reduce the constraint pattern bodies to cache optimized code specific to the variable types. Code customized in this way can take advantage of the type knowledge to compile direct offsets to object fields, avoiding runtime lookups.

In general, the technique of caching special-purpose objects created from general constraint patterns can be very useful for improving performance; see section 5.4.4 for a discussion on how such patterns can serve as function caches for incremental computation mechanism.

5.3.3 User-level Hints to the System

We can improve system performance by allowing the user to give hints concerning the use of certain objects. Some objects will play a particular role in the program, and can be optimized in that role. These objects include *constant* objects, which will never change, and *read-only* objects, which change value but cannot be modified by external clients. Also, it may be the case the user will know the likely access patterns of a particular object and want to optimize for these

access patterns explicitly. To do this, we would like to tell the system that certain attributes should be considered as state, while other attributes should be computed as functions of other parts of the state.

Constant and ReadOnly Objects

We can use Siri's multiple inheritance mechanism to annotate objects as `Constant` or `ReadOnly`. These annotations are simply additional prefixes that do not change the basic attributes or constraints of the object, but instead provide special information to the solver.

A *constant* object in the system has the property that once it has a value it will never be changed. Instances of objects already treated as constant include literal strings and numbers, and we can often code basic system patterns as constants. This can be thought of as a form of extended open-coding (see [Lang86]), where the codes inlined to a compiled object are any `constant` object. Since these objects are immutable, they can be shared freely without concern for updating, and specifically, we do not need to maintain pointers to dependents in case the object changes state, saving space in the object's dependency list. In addition, we save time by being able to inline object values throughout the system.

For example, we can define a constant object `wordSize` that is used to specify the word size of the host machine as follows:

```
wordSize: anInteger, Constant { self = 32; };
```

Here, `Constant` is used as a mix-in object solely for the system's use for optimization. This prefix indicates to the system that it can inline instances of `wordSize` freely into other objects, since `wordSize` will never change.

We can annotate other objects as *read-only* objects. Trivial instances of objects already treated as read-only include label parameters that are bound when the label is matched to an expression. By annotating certain objects as read-only, the system can compile code that takes all read-only objects, both system-defined and user-annotated, as input parameters to a satisfaction method. Annotating objects as read-only tells the system that the objects may not be modified in the evaluation of the satisfaction method, but must be bound before invoking it. Read-only objects are similar to objects of `known` type in Bertrand [Leler88]; the process of compiling lambda expressions for satisfaction methods in Bertrand uses objects of this kind.

While a read-only annotation can be considered as a convenience or hint to the solver, there are circumstances when an object *must* be read-only. An example of a read-only object is the `currentDate`:

```
currentDate: aDate, ReadOnly { --code to return the current date-- };
```

Since the `currentDate` does change, it is not a constant, but any code that references `currentDate` may not modify it.

Storing State Explicitly

Finally, often the different attributes of an object are not equally important to clients. For example, we may know that in our particular application, Kelvin temperatures are used almost exclusively, with occasional use of Celsius and Fahrenheit. If the relationships are simple enough, one can optimize the definition to trade off space or time in each temperature object. This optimization knowledge is localized in the definition of the temperature object, where it belongs, rather than distributed throughout the system as conversion functions.

As with the `Constant` and `ReadOnly` annotations, we may annotate objects to be `Stored` by providing an additional prefix in the constraint pattern. For example, the temperature definition

```
aKCFTemp: {
   K, C, F: aNumber, Stored;  -- Known to be stored as state
   F - 32 = 9/5 * C;
   K = C + 273;
};
```

stores the attributes directly as state, and creates satisfaction methods to compute each attribute in terms of the others. Alternatively, we can store only the Kelvin value as state:

```
aKCFTemp: {
   K: aNumber, Stored;
   C, F: aNumber;
   F - 32 = 9/5 * C;
   K = C + 273;
};
```

and have shared satisfaction methods for changing that value and returning each of the three different views.

5.3.4 Compilation

Siri currently evaluates objects by reducing them using constraint patterns. These patterns simplify expressions and solve for variable values when possible, but do not actually generate machine code for evaluation. Object-level methods can be compiled to machine code, however, by setting up constraints that solve for attributes based on the known method parameters. A satisfaction method generated by the system to compute an attribute value can also be compiled to machine code in the same way.

We can write the code to perform compilation directly in Siri using constraint patterns as well, using the existing mechanisms for manipulating expressions. In fact, we can use the same strategy that Bertrand uses: the introduction of a special type of object, called a *lambda*, that represents a functional expression that takes a constant as an argument and evaluates to a constant. This expression is evaluated via an *apply* operator that takes a lambda expression and its parameters and returns the result. Complicated expressions can be rewritten to primitive lambda expressions defining arithmetic and access to attribute values. These primitives can be threaded interpreted code, or can actually return expressions representing machine code to be evaluated when the method is invoked. All of these new objects can be directly coded in Siri.

For more details on compilation techniques using augmented term rewriting, see [Leler88].

5.4 Incremental Recomputation Techniques

Siri maintains object dependencies like many other constraint systems; for performance reasons, we would like to update as small a piece of the dependency network as possible. The system should notify objects of changes only when necessary; if an object is reevaluated with new data and its state remains unchanged, it should not notify dependents. To minimize this notification process, we can use an algorithm such as incremental attribute propagation [Hudson91]. Using such an algorithm, we can limit the propagation of new values to only those objects that actually need them for resatisfaction.

However, there is another way to minimize recomputation. In many systems, such as word processors, graphics editors, or programming environments [Reps87, Robison87, Pugh88], the user performs editing operations on a large model that must be resatisfied incrementally (to be redisplayed, for example). In this resatisfaction process, many computations that had been performed previously are performed again with the same values, since only a small part of the model has been changed. We would like to be able to shortcut these redundant computations; if a function is invoked multiple times with the same parameters, we would like to take advantage of this fact.

In addition, many program optimization techniques that are coded by programmers depend on caching information, knowing when information needs to be recomputed, and when it doesn't. By providing a mechanism by which this can be determined by the system, rather than by a programmer's intimate understanding of program dependencies, efficiency can be gained safely while disallowing situations that could produce inconsistent data.

The mechanism that we consider using to address these issues is called *incremental recomputation*. Incremental recomputation is an idea based on the assumption that subsequent evaluations of the model will be substantially similar to previous evaluations. In such cases, if we have programmed parts of our system in a functional style (without side-effects) we can shortcut the evaluation process by caching the results of previous function calls. Then, instead of reevaluating functions every time, we may be able to look up the result in the cache, keyed by

the function's parameters. This concept is called *function caching*, or *memoizing* [Michie68].

In order to perform function caching effectively, we need to solve several problems. First, we need to be able to find the function's cache entries quickly, based on the function name and its arguments. Next, we need a way to test for object equality very quickly, so that we can determine whether a function's arguments have been seen before; if so, we can return the cached result. We need a way to determine the functions for which caching is beneficial. Also, we need useful policies for maintaining the function caches, in particular, purging and replacement policies. Finally, we need to decompose problems such that solutions to subproblems may be cached and reused efficiently by sharing.

Pugh developed mechanisms mechanisms for the efficient incremental recomputation of functional programs using function caching in his Ph.D. thesis [Pugh88]. In particular, he developed data structures, algorithms, and heuristics for solving the above-mentioned problems. In his thesis, he discusses techniques for solving each of the problems described previously. Some of his most interesting results include an algorithm called *lazy structure sharing*, which is an amortized algorithm for speeding up a standard method for testing equality, a recursive subtree walk. In addition, he developed other amortized algorithms to test for equality on lists and sets, object aggregates for which object equality testing is normally quite expensive.

We can use some of Pugh's results directly, such as his hash function specification for mapping a function call instance to a hash table address. Since Siri is not a purely functional language, we will have to modify some of the approaches that Pugh used to adapt to Siri's needs. The following sections discuss the issues of object equality, lazy structure sharing, and object decompositions in the context of a Siri implementation.

5.4.1 Object Equality

In Siri, we could use Pugh's techniques for an incremental recomputation facility. While this would not increase Siri's performance in an absolute sense, it would improve the speed of interaction in applications that involve incrementally editing parts of large models. However, there is a problem: Siri's constraint pattern model does not define object equality, a necessary part of incremental recomputation. Equality in Siri is a notion that is defined by the programmer, not the system (however, equality of system-defined objects such as strings and integers are in fact defined by the system). Object equality is simply a relation between objects of the same type, or of different types, and is defined by the programmer with a constraint pattern. For example, say for some reason we define equality between two rectangles as follows:

```
"r1'aRectangle = r2'aRectangle": aBoolean {
   r1.width = r2.width & r1.height = r2.height;
};
```

Two rectangles could be equal even though their attributes `top`, `left`, `bottom`, and `right` do not correspond. In the general case, testing for equality between objects can be quite expensive,

requiring that a user-defined pattern be evaluated (in fact, this is one of the reasons for a particular restriction on labels: no parameter may appear twice in a label, otherwise an equality test must be performed). Since we require the ability to test for equality at the system level, we need a way to do this conservatively without having to actually evaluate equality patterns. It is not necessary to determine equality in every single case, but we should be able to in enough cases that incremental computation will actually be usable. Therefore, we must define object equality in such a way that system equality testing is invisible to the user, but still performs adequately well so that function caching can pay off. The conservative checking approach simply fails to take advantage of all possible cache entries; it does not change the semantics of the incremental recomputation.

There is an additional complication: because Siri does allow modification of objects, unlike functional languages, we cannot simply test for equality between two objects by testing their pointers. Instead, each object must maintain a modification value that is incremented whenever the object changes.

Equality testing starts out very simply[1]. First, we test for the pointer equality of the objects; if the pointers are the same, then the object's modification value is checked against the saved value in the cache. If they are the same, then the object is the same and has not been modified. This is the most common check; we expect that in the general case of checking two objects with different pointers, they will most often turn out to be not equal.

If the pointers are unequal, the objects to which they refer may still represent equal values. In this case we will perform a conservative check without evaluating any user-level equality patterns. First, we check to see if the lengths of the objects are equal; if not, we assume inequality. The types of the objects must also match (their prefixes and families); if not, again we assume inequality. If these two special checks succeed, then the system can go on to the general equality testing process.

The general equality test proceeds as follows. The basic assumption is that structural equality is a more stringent criterion for equality than a user-level pattern. For example, the two points `(1, 2)` and `(1.0, 2.0)` are equal, but would not be found to be structurally equivalent. We use this assumption to perform the conservative check. The system performs a recursive tree-walk of the object's subparts. We require that the subparts of each object be recursively equal; if they ever fail to be equal, the checking process fails. We can also limit the level of recursive checking and fail early.

This test may not perform well in practice initially. Instrumentation of a running system will help determine the bottlenecks causing the poor performance; we may then have to define less conservative tests, or special tests for particular object types, for acceptable performance.

[1]We in fact test for inequality, since there are many levels at which an equality test may fail.

5.4.2 Lazy Structure Sharing

We can shortcut the recursive tree-walk procedure and determine equality more quickly through Pugh's *lazy structure sharing* technique. The tree-walk procedure walks down the subtrees of each object, and at each node, checks for equality. If at any point two subtrees are determined to be equal, one may be discarded and replaced with a pointer to the other with a `copy-on-write` reference. The next time that the tree-walk procedure is performed, a simple pointer test suffices to determine equality, and the recursion can stop at that point.

Since Siri is not a strictly functional language, there must be a safeguard against modification of these shared structures. Since the structure is shared with a `copy-on-write` reference, the mutator will perform a copy of the structure in order to preserve ownership of the private structure whenever it is modified.

5.4.3 Object Decompositions

Some aggregate objects, such as lists and sets, are difficult to test for equality quickly; we need a way to decompose such objects recursively so that large pieces can be checked at once, thereby reducing the equality testing time. Normally, equality of lists and sets is tested by checking each of their corresponding entries for equality. By structuring a list through Pugh's *chunky list decomposition* we decompose it into pieces, each containing many entries. We then check for equality quickly through a recursive descent of its parts, as when checking normal structured objects. We use special identifiers in each chunk, like object modification numbers, to determine chunk equality quickly, and carefully code `insert` and `delete` operations to maintain the list structures, chunks, and modification number invariants.

5.4.4 Pattern Caching

Given mechanisms for fast equality testing, and list and set objects designed for decomposition, we can build an incremental recomputation facility. In Siri, we can think of non-side-effecting constraint patterns as functions, and we can cache the result of instantiating such a constraint pattern in the same way as we cache the result of a function call.

As in pattern customization (section 5.3.2), we can create special constraint patterns to provide additional information for shortcutting the evaluation process. However, instead of replacing the types annotating parameters in a constraint pattern label with more specific types, we can replace the parameters themselves with specific objects. These customized patterns thus serve as function caches that match previously-computed results and return them directly. This approach benefits from using the existing term rewriting system for looking up matching parameters in the function cache, thus avoiding the need for an additional mechanism.

When the system evaluates a constraint pattern with a set of parameters, it can either return the result, or return the result and in addition save a customized constraint pattern with the result as a cache entry. To create an entry, the system makes a new constraint pattern whose label is

the original pattern's label with the actual parameter values in place, and whose residual is simply the result of the evaluation. Thereafter, the pattern matcher will automatically match the more specific label first when possible (e.g., the label with the value parameters) and return the computed value directly. For example, the constraint pattern for creating an instance of `aPoint`,

```
"a'aNumber pt b'aNumber": aPoint { x = a; y = b; self };
```

can be instantiated with `a = 2, b = 3`, to the point `(2, 3)`. We can then cache a customized version of the pattern based on these parameters as

```
"2 pt 3": aPoint { x = 2; y = 3; self };
```

which can be compiled to the point instance. Matching the expression `2 pt 3` returns directly an instance of the point, avoiding a runtime evaluation of the pattern.

These caches only work as long as the values in the label do not change; they must either be constant values or objects that have not been modified since the last invocation. The fast equality testing mechanism is used to ensure that the full state of the parameters being matched is unchanged from the time the customized pattern was first created and evaluated.

As with pattern customization, such object patterns must be maintained separately from the user-defined patterns, so that the customized patterns can be purged when necessary and so that they are not mistaken for user code. Pugh addresses issues such as the size of the cache, what to add to the cache, as well as what to purge and when, though the issues would be different in a non-functional system such as Siri with modifiable objects. In Siri, we would have to purge objects with cache entries as soon as they are modified, since they would no longer have the same value as they did when first cached.

5.4.5 Evolutionary Reduction

Since Siri's reduction process performs both interpretation and compilation, we can take advantage of this continuum by creating customized patterns as they are encountered. As objects are evaluated, we can cache their results as customized patterns; as objects change, we may further reduce the cached pattern values as additional information, such as values of external objects, becomes available or changes. Like SELF [Chambers89], we can create customized objects by inlining the values of other objects even if they are not constant. When such objects do change, we can use a dependency mechanism to revert the customized object back to its general form.

We could also create mechanisms for watching the attribute access patterns. By watching these patterns we can deduce which objects are read and written most often, and thus should be stored as state, and which should be coded as satisfaction methods to save space. In this way the system can provide its own `Variable` and `Stored` hints for improving the storage structure and

evaluation performance. Because of the computational requirements of these self-observing mechanisms, we might use this process primarily during development, as a way for the system to observe its own behavior and suggest optimizations to the programmer.

5.5 Solver Optimizations

We now consider various techniques for improving the ability of the constraint solver in Siri. Since the constraint solver is written in Siri, it is (in principle) extensible by the user. However, in practice it is preferable to have a more powerful solver available so that users need not be concerned with issues that they had hoped to avoid by using Siri in the first place.

We can improve the Siri solver in several ways: the solver itself can be extended; we can add domain-specific solvers as is done in CLP(R); and we can reimplement Siri to maintain a constraint network as is done in systems such as ThingLab and Kaleidoscope.

5.5.1 Extending the Solver

We can extend the existing solver in simple ways to improve its power. One technique is to introduce variables to simplify complex expressions, making them more easily solvable by the constraint system [Michaylov92]. We can use a two-pass process to introduce new variables; these variables replace subexpressions of the original expression being evaluated. These subexpressions, by appearing with their own variables, can be solved for separately or delayed in evaluation until enough other variables are known. For example, the expression

```
a = 2*b + 3*c;
```

can be rewritten into the set of expressions

```
a = r + s; r = 2*b; s = 3*c;
```

and `r` and `s` could be solved for separately. This can help in evaluation of slightly non-linear expressions. We can take out the non-linear terms as variables; this may allow the main expression to be solved linearly. As other terms become ground, the likelihood that non-linear terms can be reduced to linear ones becomes greater.

We can use the incremental computation engine to implement the variable introduction technique. In particular, we can use the equality testing and structure sharing mechanisms to detect and rewrite common subexpressions. For example, we can use the equality testing machinery in the two equations

```
a = 4*r + s; b = 4*r + t;
```

Instead of performing the conservative test which would fail immediately (since the variable `a` is not the variable `b`), we can force the equality testing procedure to proceed through the complete

expression tree. In this process, the procedure can detect the common subexpression `4*r`. Normally, one occurrence would be replaced by a pointer by using lazy structure sharing. In this case, we can instead take the subexpression out from each expression and equate it to a variable binding `q`, replacing the subexpressions with a variable rather than a pointer:

```
a = q + s; b = q + t; q = 4*r;
```

This approach can also improve performance in addition to helping the solver. In particular, factoring common subexpressions is similar to jungle evaluation, in that the subexpression `4*r` need only be evaluated once when it is shared. Generally, the greater the sharing, the greater the performance benefit.

5.5.2 Domain-Specific Solvers

A more ambitious scheme for improving Siri's constraint satisfaction ability would be to provide a set of domain-specific solvers. This approach is used in CLP(R) [Heintze87, Michaylov92] quite successfully. For example, we could add to the simple linear solver a numeric solver like Newton-Raphson for nonlinear expressions, and a Simplex solver for handling inequalities. In addition, considering each Siri object as a complete CLP(R) system for constraint satisfaction purposes (excluding the Prolog semantics) might be a useful way to take advantage of many of the features of CLP(R) and thus improve Siri's constraint engine.

For example, using specially-typed operators and an ordered normal form similar to the one used in ordered linear combinations, we can recognize equations such as

```
a*x + b*y + c*z < 15; h*x + r*y + n*z < 20;
```

as being solvable using the simplex algorithm. Instead of manipulating expressions equated to zero, we manipulate inequality expressions whose root operator is a special subtype of the < operator. The term rewriter can then recognize and manipulate these expressions by matching on the special operator type, and package them up into special objects whose solutions are provided by the simplex solver.

5.5.3 Using a Constraint Network

A system such as Kaleidoscope does not manipulate expressions symbolically; instead, it compiles complex expressions into networks of primitive constraints that are solved by value propagation. This is quite similar to the variable introduction idea of section 5.5.1 For example, the Fahrenheit-Celsius conversion equation

```
F = 9/5 * C + 32;
```

can be rewritten as three primitive constraints:

```
a = F - 32; b = 9/5 * C; a = b;
```

With the two unnecessary variables removed, these three constraints represent the graph of primitive constraint nodes

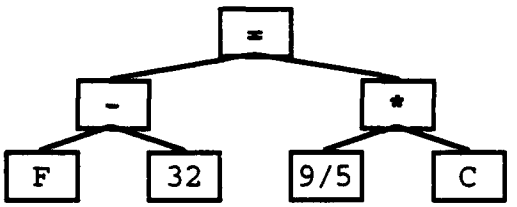


Figure 5-1. The constraint graph for `F - 32 = 9/5 * C`.

Each of the primitive binary nodes can solve for the value of one of nodes to which it is connected in terms of the other two. (The equality node simply passes its value through from one side to another.) For example, the `*` node can solve for `X` given `Z` and `Y`, `Y` given `X` and `Z`, and `Z` given `X` and `Y`, depending on which way the data flows:

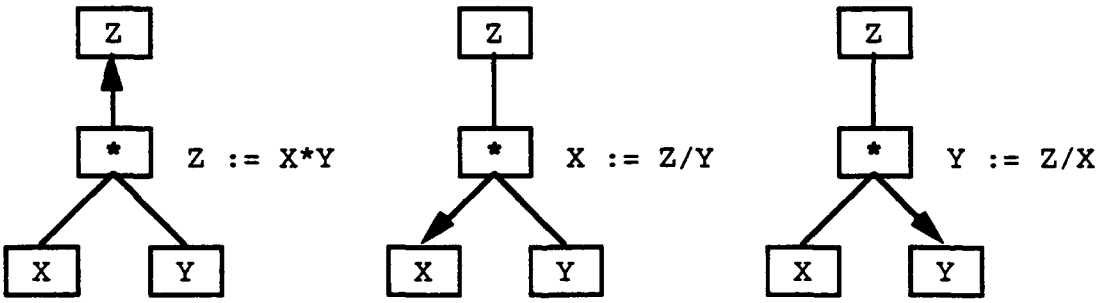


Figure 5-2. The three data paths for `X * Y = Z`.

Thus, given `F` we can derive `C` by accumulating the appropriate satisfaction functions on the path from `F` to `C`:

```
(F-32)/(9/5) = C;
```

One of the benefits of this organization is that we can add and remove constraints from the network easily, and we can propagate values very quickly through the network to solve for variable values. In particular, by using an algorithm such as DeltaBlue [Freeman-Benson89], we can do this with weighted constraints so that certain constraints are considered more important than others. Weighted constraints allows for the definition of default behavior and control of the satisfaction process.

We could use some of these ideas in Siri as well. Instead of using term rewriting to reduce objects to residuals, we could compile them to a network of primitive constraints and use an algorithm such as DeltaBlue to update the network and propagate values for constraint satisfaction. Using such an organization would have several benefits:

- We could satisfy large sets of constraints more quickly, since local propagation is extremely fast.
- Constraints could be weighted, for easier specification of default behavior.
- We could add and remove constraints quickly, so rather than having a single set of temporary constraints within a method that are active during the entire method evaluation, we could have constraints that are live during a particular sequence of operations.
- We could have a less strict interface to objects, and allow arbitrary attributes to be modified directly instead of through a method interface.
- We could take advantage of others' experience with DeltaBlue and related solvers to handle cases that Siri currently does not handle, such as linear inequalities, and to compile plans for repeated satisfaction.

However:

- DeltaBlue is optimized for large networks of constraints. Since Siri objects typically contain few constraints, the benefits of being able to handle large number of constraints is not as important.
- Value propagation cannot handle simultaneous equations.
- Allowing direct modification of attributes, though flexible, breaks encapsulation and makes the abstraction much weaker.

Still, it would be interesting to implement a version of Siri that provided constraint networks for constraint satisfaction within encapsulated objects. Perhaps we could use a hybrid model, where we use term rewriting in a more limited role to simplify portions the constraint network to handle symbolic constraints, and to manipulate and reduce the equations to primitive terms; and use DeltaBlue to handle the traditional local propagation constraint satisfaction process.

5.6 Summary of Optimizations

There is a wide variety of techniques available for improving the current implementation of Siri. We can use straightforward, known approaches for improving Siri's code density by

factoring out common information; for improving its speed through optimization of the term rewriting process and through compilation of satisfaction methods; for avoiding redundant computation through function caching strategies; and for improving Siri's constraint solver.

Some of these approaches work especially well in combination. The incremental computation mechanism can be used for a variety of purposes, including detection of common expressions that can be factored out to reduce rewriting, and for introduction of variables.

The success of these techniques in other systems gives evidence of their applicability and usefulness. We hope to try some of these techniques in future versions of Siri.

Chapter 6
Summary of Contributions and Future Work

There are many reasons to design and develop a programming language. We hope that the ideas in this thesis will act as catalysts for newer and better programming languages; that the constraint pattern concept will help people think of ways to extend object-oriented programming; and that the system itself could be useful for experimental programming.

In the following sections we will summarize the contributions of this thesis, and describe some ideas for future work with Siri.

6.1 Contributions

This thesis addresses some existing deficiencies in object-oriented programming. First, in traditional object-oriented programming, programmers are not able to encode models of consistency requirements within objects; instead, they must write individual methods with these consistency requirements in mind, thus threading implicit information throughout the object's definition. Also, method definitions depend heavily on an object's implementation; in particular, method code must handle computed attributes, or functions of the object's state, differently than instance variables, which are purely stored state. Any changes to the organization of the object, such as making attributes into stored instance variables, or instance variables into computed attributes, requires that each method that accesses this information be recoded to reflect this. Finally, using inheritance in object-oriented programming requires that the programmer know and understand the implementation of the objects being inherited; thus, there is no encapsulation of inherited objects.

The constraint programming approach provides some attractive features for solving these problems. We use equational constraints to make explicit the consistency requirements within objects, as well as for writing more abstract methods that do not require detailed knowledge of an object's implementation. Unfortunately, constraint systems generally do not provide encapsulated objects, since constraints on an object can modify its state without going through its message interface.

This thesis provides an approach for combining the best features of both object-oriented programming and constraint programming. We provide an object-oriented programming system where we can: specify consistency requirements explicitly using constraints; maintain encapsulation, so that each object is responsible for modifying its own state; support encapsulation of inherited objects; and provide explicit control of the constraint satisfaction process. We present three major contributions:

- A model of object-oriented programming based on equational constraints that provides the object-oriented features of encapsulation and inheritance.

- An execution model, ATR+, that uses term rewriting for symbolic constraint satisfaction in an object-oriented framework.

- The design and implementation of a prototype language, Siri, that provides a complete object-oriented language using a single abstraction mechanism called a *constraint pattern*.

Each of these contributions will be discussed in the following sections.

6.1.1 The Programming Model: Constrained Objects

The constrained objects programming model provides object-oriented programming with equational constraints for maintenance of internal object consistency. Constrained objects provides a two-level organization where encapsulated objects communicate among each other via message-passing, and objects within an encapsulated object communicate via constraints.

An encapsulated object consists of three parts: its *attributes*, which define the object's traits as well as its state; its *constraints*, which define what is considered to be valid states; and its *methods*, which define specific ways to perturb the object's state while maintaining consistency.

Encapsulated objects communicate via messages. Clients of an encapsulated object may send messages to examine its attributes to gain information about its state, or to invoke methods for changing its state. In this way, the encapsulated object is fully responsible for its own internal state, since no object may constrain its state externally.

Internal subobjects within an encapsulated object communicate via constraints on their attributes. Special subobjects define methods, which are objects that temporarily constrain the object in order to compute new attribute values. These methods may also contain imperative code for sending messages to external encapsulated objects.

By defining the internal communications of an encapsulated object with constraints, the model makes explicit the object's consistency requirements, which, in traditional object-oriented programming, had been implicitly encoded in each of the object's methods.

The two-level organization does have some limitations. We restrict pointer manipulation, and do not allow constraints on external objects since they can be used to bypass the object's interface. Thus we restrict constraint solving to a single encapsulated object, and do not allow arbitrary constraints in the system. However, these limitations are chosen in order to retain the important aspects of object-oriented programming: encapsulation of objects and message-passing.

6.1.2 The Execution Model: ATR+

ATR+ is an execution engine for the constrained objects model. It is an object-oriented term rewriting system that extends augmented term rewriting to include object encapsulation, inheritance, and a concept of `self`-reference. ATR+ provides nested object definitions with structured merging of objects and their constraints. We use ATR+ for evaluation, object reduction and compilation, and constraint satisfaction. Since the constraint satisfaction engine is not intrinsic in the system but evaluated by ATR+ at the user level, the execution model has no notion of constraint; this makes it possible for programmers to define their own solvers, or their own expression evaluation mechanisms.

6.1.3 Siri, the Proof-of-Concept Language

Siri is the proof-of-concept language described in this thesis that implements the constrained objects model using ATR+. In Siri, *constraint patterns* are the single abstraction mechanism in the language, subsuming classes, instance variables, methods, and control structures.

We use constraint patterns for definition and instantiation of prototypes, constraint inheritance, methods for modifying object state, and control structure evaluation. Siri implements a simple equation solver for linear and near-linear equations using special patterns; this solver can be used for constraint satisfaction, method generation and compilation. An inheritance facility based on the concept of prefixing objects provides for single and multiple inheritance in a simple and general framework.

The benefits of a language based on constraint patterns are many-fold:

- The language is uniform and simple, with a single abstraction mechanism for all object-oriented structures;
- Declarative method code is more expressive, simpler, and safer than imperative methods, by using constraints to set state values, thus maintaining the object's internal consistency;
- Constraint satisfaction through equation solving is useful for compilation and optimization;
- Abstract patterns and methods in Siri are more general and more amenable to reuse than traditional object-oriented methods that require knowledge of their objects' implementations; and
- The language can be extended with new syntaxes and solvers at the user level, making Siri a good platform for language experimentation.

6.2 Future Work

There are many different directions that we can take Siri and the constrained objects model. We already discussed in detail in Chapter 5 basic improvements in Siri's implementation; other directions include changes in the programming model to make it more general; providing concurrency; taking advantage of Siri's rule-based aspects to define active agents; and providing programming environment support.

6.2.1 Changes in the Programming Model

One of the major limitations in the constrained objects programming model is the two-level organization, in which encapsulated objects send messages, and internal subobjects are constrained.

One can imagine providing both message and constraint access to an object. However, the non-local effects of sending a message to an object which is also externally constrained would seem to conflict with the goals of object-oriented programming. In addition, implementation of such a feature would likely require a fully general facility for constraint-imperative programming as is found in Kaleidoscope, which would complicate the model significantly.

It would be interesting to explore allowing shared external objects to have constraints on their attributes, and to allow references to external objects to be changed at any time. This would require that objects resatisfy their state at times other than when a method is invoked, and would affect encapsulation. However, many constraint systems allow arbitrary constraints within the system, and it is worth investigating how important encapsulation is for program structuring.

6.2.2 Concurrent Evaluation

Since each object in the constrained objects model is an independent constraint system, we could allow individual objects to execute concurrently as active objects. Active objects are objects which execute independently of each other, and in parallel. This is a new and quite interesting area of research, and several different active object systems have been created, including Actors and Concurrent Smalltalk; results from this research could be applied to a concurrent version of Siri, since the main issues of synchronization and handling concurrent streams of messages are independent of Siri's constraint satisfaction features.

6.2.3 Constraint Patterns as Agents

One of the most interesting areas of future work is to leverage off the rule-based aspects of Siri. Although we think of Siri as an object-oriented language, where objects send and receive messages to change each other's state, one can also think of constraint patterns in Siri as *agents* that look for opportunities to evaluate expressions. From this perspective, we think of the constraint pattern

```
"a'aNumber + b'aNumber": aNumber { ...perform addition... };
```

as an agent object that opportunistically searches for expressions of the form `a+b` in a given context, and performs a particular action.

If we generalize an object's internal execution environment to a *working memory,* we can think of prefixes as groups of agents that define a rule base for modifying the working memory. In Siri, we currently provide built-in agent groups (e.g., sets of constraint patterns) that work together to perform constraint satisfaction. These agent groups just read and write information in the working memory, cooperating on the reduction of the equations in the common database, each agent providing its own special knowledge. We can imagine that other agent groups could be defined for evaluation of a variety of non-numeric expressions, or handle tuple-space-like associative searches. Methods could be generalized to objects that temporarily make a set of agents active within a given working memory.

To take advantage of this agent-based approach, we would also want to add metaprogramming facilities so that we can add, change, and remove prefixes (agent collections) from objects (working memories); add, change, and remove attributes of an object; and add, change, and remove constraints within the working memory. When the working memory changes, agents would notice the change and perform actions (reductions) based on the new information. In order to make this feasible, we would structure the insides of an object more strongly, so that we can directly address individual constraints by name for assertion, change or removal.

Once we have metaprogramming facilities, writing Siri programs that create new Siri programs and modify existing ones would be possible. Metaprogramming would make the creation of user-interface tools for creation and editing of dialog boxes, windows, and so on quite easy: one could simply create a description of the dialog directly in Siri using metaprogramming to specify the dialog's subparts and how they are related to each other.

6.2.4 Exploratory Programming

Finally, constraints have been used in user interface systems to provide mechanisms for updating dependency networks, and for graphical layout. One of the goals of the constraint pattern work was to explore the construction of document-based user interfaces using objects and integrated constraints. Siri could be used as a testbed for an experimental user interface system, where the uniform object model provided by constraint patterns will allow objects from buttons to documents to share the same namespace and evaluation mechanisms. New expression domains, such as region arithmetic for graphic object updating and window occlusions will be investigated, to make use of Siri's expression manipulation ability to more directly support the user interface implementation.

6.3 Conclusion

The constrained objects programming model, provides a new way of looking at a rather old programming paradigm: object-oriented programming. By making it possible for a programmer to state more explicitly the actual modeling specifications in the design of an object and its internal subobjects, the constrained objects model helps the programmer to describe more fully the intent and structure of object-oriented programs. By making the implicit knowledge of an object's consistency requirements explicit through constraints, the model improves reusability; the specification is more abstract, and thus more able to be used in different contexts. The constrained objects model provides these advantages without losing any advantages of traditional object-oriented programming.

The Siri programming language is intended to provide an example of the constrained objects model in as simple a language as possible. Siri's single constraint pattern abstraction is sufficient for describing a wide variety of object-oriented constructs. We believe that Siri is evidence that object-oriented programming languages need not be as complicated as they are; a small number of powerful constructs can do the job just as well, and perhaps more elegantly.

Appendix A: Examples

A Stack

The first example is a stack parameterized by the type of entry in the stack. The stack elements are held in a list, and the list is modified by the stack methods `push` **and** `pop`**. Attempting to pop off too many entries will result in the constraint** `depth>0` **being violated in the** `pop` **method.**

```
"stack aT'aType": {
   depth: aNumber;
   elements: aList;

   "initially": aMethod { depth = 0; elements = nil; };

   "top": aT { elements.head };

   "push x'aT": aMethod {
      depth = previous depth + 1 !
      elements = x, previous elements; };     --Comma is list constructor

   "pop": aMethod {
      depth>0;
      elements = previous elements.tail !
      depth = previous depth - 1; };

   "empty": aBoolean { depth = 0 };

   self
};
```

Another way to write the stack is to define `elements` **as an indexed object. The** `depth` **of the stack is maintained, as before, but is also a parameter to the elements definition stating the size of the array. When** `depth` **changes,** `elements` **changes size as well.**

```
"stack aT'aType": {
   depth: aNumber;
   elements@(1 to depth): aT;

   "initially": aMethod { depth = 0; };
   "top": aT            { elements@depth };
```

```
   "push x'aT": aMethod {
      depth = previous depth + 1 ! elements@depth = x; };

   "pop": aMethod {
      depth > 0; depth = previous depth - 1; };

   "empty": aBoolean { depth = 0 };

   self
}
```

Factorial

Factorial is a traditional functional programming example. Here we define it by cases within a single object. The internal definition of fact shadows the external name; the recursion occurs within the object.

```
"fact n'aNumber": aNumber {
   "fact 1": aNumber { 1 };
   "fact n'aNumber": aNumber { n * fact (n-1) };
   fact n };
```

We can also write factorial using a single if expression to handle the base case of `n = 1`.

```
"fact n'aNumber": aNumber {
   if (n = 1) then 1 else n * fact (n-1) };
```

Finally, we can define factorial as a product of a list of integers, writing patterns that create the list and patterns that consume it.

```
"ints 1":  aNumber { 1 };
"ints n'aConstant": aList { (ints n-1), n };

"prod n'aConstant": aNumber { n };
"prod(a, b)" : aNumber { (prod a) * (prod b) };

"lfact n'aConstant": aNumber { prod(ints(n)) };
```

Temperature Conversion

No thesis dealing with constraints would be complete without a temperature conversion example. Here we define two objects, one describing the relationship between Fahrenheit and Celsius, and the other extending the first to include Kelvin.

```
aCFTemp: {
   C, F: aNumber;
   F = 9/5*C + 32;
};

aKCFTemp: aCFTemp {
   K: aNumber;
   K = C+273;
};
```

Electrical Components

Electrical circuits are similarly in vogue among constraint language theses. Here we implement the electrical circuit simulator from [Freeman-Benson91].

```
anElectricalComponent: anObject;
```

An electrical lead is an electrical component with two attributes: the potential and the current.

```
anElectricalLead: anElectricalComponent {
   potential, current: aNumber;
   self
};
```

A solder joint has a number of wires hooked into it. Each wire is an electrical lead with a potential and a current. The sum of all of the currents must be zero, and the potentials must all be the same by Kirchoff's laws. We use a map over all of the wires to create an expression equating the sum of the currents to zero, and the potentials to each other.

```
aSolderJoint: {
   wireCount: aNumber;
   wires@(1 to wireCount): anElectricalLead;

   zeroSum: { wires@(1 to wireCount) map { each.current + ... } = 0 };
```

```
   equalPotential: {
      if (wireCount>1) then (
         base = wires@1.potential;
         wires@(2 to wireCount) map { each.potential = base; ... };
      );
   };

   zeroSum; equalPotential; self
};
```

We can connect leads to each other, making a solder joint. All we have to do in this case is set the wire count and the wires to the two leads.

```
"lead1'anElectricalLead connectTo lead2'anElectricalLead": aSolderJoint {
   wireCount = 2;
   wires@1 = lead1; wires@2 = lead2;
   self
};
```

Similarly, we can make a three-way connection by setting `wireCount` to 3 and equating the first three elements in the wires array to the three leads.

```
"lead1'anElectricalLead connectTo lead2'anElectricalLead
            connectTo lead3'anElectricalLead": aSolderJoint {
   wireCount = 3;
   wires@1 = lead1; wires@2 = lead2; wires@3 = lead3;
   self
};
```

Now we define the various two-leaded components: a battery, a resistor, and a ground.

```
aTwoLeadComponent: anElectricalComponent {
   lead1, lead2: anElectricalLead;
   lead1.current = -lead2.current;
   self
};

"aBattery v'aNumber": aTwoLeadComponent {
   voltage: aNumber;
   voltage = v;
   lead1.potential = lead2.potential + voltage;
   self
};
```

```
"aResistor o'aNumber": aTwoLeadComponent {
   resistance: aNumber;
   resistance = o;
   lead1.potential - lead2.potential = lead1.current * resistance;
   self
};

aGround: anElectricalComponent {
   lead1.potential = 0;
   self
};
```

A switch is slightly different: the constraints that are maintained depends on the state of its `active` attribute. When active changes, the switch, and all of the objects of which it is a part, must be reevaluated.

```
aSwitch: aTwoLeadComponent {
   active: aBoolean;

   if active then
      lead1.potential = lead2.potential
   else
      lead1.current = 0;
};
```

Now we can build sample circuits, hooking up a variety of components and requesting the attributes of the various leads. The next two circuits are completely static, and thus are evaluated only once.

```
example1: {
      b: aBattery 10;
      r: aResistor 5;
      g: aGround;

      b.lead1 connectTo r.lead1;
      b.lead2 connectTo r.lead2 connectTo g.lead1;
      b.lead1.current
};

example2: {
      b:  aBattery 10;
      r1: aResistor 3; r2: aResistor 7; r3: aResistor 2.9;
      g:  aGround;
```

```
      b.lead1  connectTo r1.lead1 connectTo r2.lead1;
      r1.lead2 connectTo r2.lead2 connectTo r3.lead1;
      b.lead2  connectTo r3.lead2 connectTo g.lead1;
      b.lead1.current
};
```

The third circuit includes a switch; in this case, when the switch becomes open or closed, the object must be reevaluated to take into account the change in active constraints. This is the kind of example in which incremental recomputation, or an open and modifiable constraint network, would prove useful.

```
example3: {
      b: aBattery 10;
      r: aResistor 5;
      g: aGround;
      s: aSwitch;

      open: aMethod  { s.active = false; };
      close: aMethod { s.active = true; };

      b.lead1 connect s.lead1;
      s.lead2 connect r.lead1;
      b.lead2 connect r.lead2 connect g.lead1;
};

testExample3: aMethod {
      test: example3;

      test.open ! print test.b.lead1.current !
      test.close ! print test.b.lead1.current;
};
```

A Scroll Bar

Here we define a traditional Macintosh-style scroll bar, with a fixed-size thumb. Each invocation of a method determines a state change; thus the values of the scroll bar's attributes are recomputed each `incValue`, `decValue`, **and** `setThumbTo`.

```
aScrollBar: {
   outside, up, dn, thumb: aRectangle;
   min, max, value: aNumber;
```

```
initially: aMethod {
   outside = rectangle (0, 0, 100, 20);
   (min, max, value) map { each = 0; ... };
};

-- Utility routines, used inside aScrollBar
#"a'aNumber pinnedTo (min'aNumber, max'aNumber)": aNumber {
   if a<min then min else
      if a>max then max else
         a
};

#"rect'aRectangle contains pt'aPoint": aBoolean {
   pt.x >= rect.left & pt.x <= rect.right &
   pt.y >= rect.top  & pt.y <= rect.bottom
};

#inUp:      aBoolean { mouse.down & up contains mouse.location };
#inDn:      aBoolean { mouse.down & dn contains mouse.location };
#inThumb:   aBoolean { mouse.down & thumb contains mouse.location };
#inOutside: aBoolean { mouse.down & outside contains mouse.location };

#incValue: aMethod { value = previous value + 1; };
#decValue: aMethod { value = previous value - 1; };
#"setThumbTo t'aNumber": aMethod {
   thumb.top = t pinnedTo (outside.top, outside.bottom-thumb.height);
};

"setRect r'aRectangle": aMethod {
   min, max, value fixed;
   outside = r;
};

mouseDown: aMethod {
   min, max, outside fixed;

   if inUp then while inUp do incValue
   else     if inDn then while inDn do decValue
   else     if inThumb then (
      vOffset: aNumber;
      vOffset = mouse.location.y - previous thumb.top;

      while (inThumb & inOutside) do (
         setThumbTo mouse.location.y - vOffset
```

```
        );
      );
   };

   "displayOn d'aDisplay": aMethod {
      d.( setFill gray ! drawRect outside ! setFill white !
            drawRect up ! drawRect down ! drawRect thumb);
   };

   (up, down, thumb) map {
      each.width = outside.width;
      each.left  = outside.left;
      each.right = outside.right;
      each.square; ...};

   up.top = outside.top; down.bottom = outside.bottom;

   thumb.top = outside.top +
               ((value-min)/(max-min))* (outside.height-thumb.height);
};
```

Rational Number Objects and Operations

We can use Siri's facilities for matching polymorphic expressions to create and manipulate rational numbers. A conversion routine, `asRational`, is used to convert integers to rationals; other routines can be defined for other numeric types. For example, `aFloat` might use continued fractions to determine a rational approximation.

```
-- Creation

aRational: {
   num, denom: anInteger;
   gcd num denom = 1;
   denom ~= 0; self
};

"a'anInteger / b'anInteger": aRational {
   num = a; denom = b; self reduced };

-- Mixed type arithmetic

"a'anInteger + b'aRational": aRational { a asRational + b };
"a'anInteger - b'aRational": aRational { a asRational - b };
```

```
"a'aRational - b'anInteger": aRational { a - b asRational };
"a'anInteger * b'aRational": aRational { a asRational * b };
"a'anInteger / b'aRational": aRational { a asRational / b };
"a'aRational / b'anInteger": aRational { a / b asRational };
"a'aRational = b'anInteger": aBoolean  { a = b asRational };
"a'aRational > b'anInteger": aBoolean  { a > b asRational };

-- Rational arithmetic

"a'aRational + b'aRational": aRational {
   num = a.num*b.denom + a.denom*b.num;
   denom = a.denom*b.denom;
   self reduced };

"a'aRational - b'aRational": aRational {
   num = a.num*b.denom - a.denom*b.num;
   denom = a.denom*b.denom;
   self reduced };

"a'aRational * b'aRational": aRational {
   num = a.num*b.num;
   denom = a.denom*b.denom;
   self reduced };

"a'aRational / b'aRational": aRational {
   num = a.num*b.denom;
   denom = a.denom*b.num;
   self reduced };

"a'aRational = b'aRational": aBoolean {
   a.num = b.num & a.denom = b.denom
};

"a'aRational > b'aRational": aBoolean {
   a.num*b.denom > b.num*a.denom };

"a'aRational reduced": aNumber {
   d: anInteger;

   d = gcd a.num a.denom;
   num = a.num/d;
   denom = a.denom/d;
   if denom = 1 then b.num else b      -- either rational or integer result
};
```

```
-- Conversion patterns

"a'integer asRational": aRational {
   num = a; denom = 1; self };

"a'aRational asFloat": aFloat {
   a.num asFloat/a.denom asFloat };
```

HyperFax

Hyperfax is an experimental user interface in which documents appear as rectangular objects in three dimensional space. An observer sees these objects as "points of light," and may move them in x and y, or in z, by dragging them with the mouse in one of the two modes.

This is a demonstration of how one might program the Hyperfax application in Siri. It is basic, and only allows the display of rectangular documents in a window, allowing them to move in either Z or in X and Y. It is assumed that an outside shell would call these routines to provide the interface; these routines simply display the documents as points of light.

A Document

A document is an object with several attributes.

```
aDocument: {
   title:      aString;          -- Document title
   author:     aString;          -- Document author
   keys:       aList;            -- A list of keywords for search
   date:       aDate;            -- Creation date
   contents:   aString;          -- Document contents
};
```

The document object pattern

Documents are stored in a pattern called `HyperfaxDocuments`. They are all labeled arbitrarily (here we use `@1`, `@2`, etc.) and are retrieved by content using the `collect` function. Note that each document is `aDocument` with each attribute being constrained to a particular value.

```
HyperfaxDocuments: {
   @1: aDocument {
         title = "The Siri Hyperfax System";
         author = "Bruce Horn";
         keys = "Hyperfax", "Siri";
```

```
        date.(month = 8; day = 1; year = 1990;)
        contents = "The Siri Hyperfax System...";
    };

    @2: aDocument {
        title = "Hfx 0.1";
        author = "Jim Morris";
        keys = "Hyperfax", "Prototype";
        date.(month = 2; day = 4; year = 1990;)
        contents = "The prototype system...";
    };

    @3: aDocument { ... };
    @4: aDocument { ... };
    ...
};
```

A Point Of Light

A point of light represents a document. It has a reference to its document, a location in three-space, a size in two dimensions, a hilight color, and a boolean value for whether it is chosen.

```
aPointOfLight: {
    document:   ^aDocument;        -- A reference to the document
    location:   a3DPoint;          -- Location in space
    size:       aPoint;            -- width and height
    hilighted:  aColor;            -- Current hilight color
    chosen:     aBoolean;          -- Currently chosen?

    initially: aMethod {
        location.(x = 0; y = 0; z = 0;)
        size.(x = 120; y = 30;)
    };

    if chosen then hilighted = black else hilighted = neutral;
};
```

An Observer

An observer has a location in three-space, a lens which determines how the view is magnified, a viewing rectangle, an origin point in the viewing rectangle, and a vanishing point. It has no constraints on its location in space, and it always is looking toward increasing Z.

```
anObserver: {
   location:   a3DPoint;          -- Location in space
   lens:       aNumber;           -- Magnification value
   view:       aRectangle;        -- A view to which objects are mapped
   origin:     aPoint;            -- Location of (0,0,0) in view
   vanish:     aPoint;            -- Location of (0,0,∞) in view
};
```

The following defines standard observer at the origin. To do this, we further constrain `anObserver` to have the location at (0,0,0), a lens value of 1000, and appropriate viewing rectangle, origin, and vanishing points.

```
aStandardObserver: anObserver {
   location.(x = 0; y = 0; z = 0;)
   lens = 1000;
   view.(
      topLeft.(x = 0; y = 0;)
      botRight.(x = 200; y = 200;)
   )
   origin.(x = 0; y = 0;)
   vanish = view midpoint;
};
```

A Point Of Light Image

A point of light image is the image that an observer sees in the viewing rectangle. It has a reference back to the point of light, a location and size of the image in the viewing rectangle, and a string title which is the same as the document title.

```
aPOLImage: aGraphicsObject {
   thePOL:     ^aPointOfLight;    -- Reference back to the point of light
   location:   a2dPoint;          -- Location of the image
   size:       a2dPoint;          -- Image size
   title:      aString;           -- Document title
};
```

The perspective transformation takes a point of light and an observer, and returns a point of light image.

```
"pol'aPointOfLight asSeenBy obs'anObserver": aPOLImage {
   D:             a3dPoint;       -- A manhattan distance from observer
   lz:            aNumber;        -- The lens parameter
   screenCenter:  aPoint;         -- The computed screen coordinate
```

```
    D = pol.location - obs.location;
    lz = obs.lens/D.z;
    screenCenter = obs.vanish - lz*(obs.vanish - obs.origin);

    -- Set the image's attributes

    thePOL = pol;
    title = pol.document.title;
    location = screenCenter+lz*D;
    size = pol.size*lz;
    self
};
```

A HyperfaxScreen

A HyperfaxScreen is an object which has an observer, a graphics window for output, and a group of documents being viewed. It creates a point of light for each document, and a point of light image in the window for each point of light.

```
HyperfaxScreen: {
    observer:       anObserver;     -- The observer
    window:         aWindow;        -- The output window
    documents:      aList;          -- The list of documents being viewed
    ptsOfLt:        aList;          -- The points of light

    -- The points of light.  Filter through the documents and
    -- create a point of light for each.

    "ptOfLightFor d'aDocument": aPointOfLight {
        document = d;
        self
    };

    ptsOfLt = documents map {
        ptOfLightFor each, ... };

    -- The window contents.  Create a POLImage for each POL.

    window.contents =  ptsOfLight map { each asSeenBy observer, ... };
};
```

A Hyperfax Controller

A `HyperfaxController` implements two subobjects which allow the user to drag the images in either `X` and `Y`, or in `Z`. Hit testing is performed in the window manager.

```
HyperfaxController: {
   "dragImage theImage'aPOLImage inXandY": aMethod {
      while theMouse.down do (
         theImage.thePOL.location.z fixed;
         theImage.location = theMouse.location;
      );
   };

   "dragImage image'aPOLImage inZ": aMethod {
      while theMouse.down do (
         image.thePOL.location.(y fixed;)
         image.location.x = theMouse.location.x;
      );
   };
};
```

An example Hyperfax display

A complete view, with a built-in arranger for `Z`, which allows the documents to be dragged around. We use a different map function called find, to select the appropriate documents. `HyperfaxDocuments`, though not a list, can be used as a list in the mapping routines.

```
JimsDocuments1: HyperfaxScreen, HyperfaxController {
   observer: anObserver;
   documents: aList;

   observer = aStandardObserver;
   documents = HyperfaxDocuments collect { doc.author = "Jim Morris" };

   pointsOfLight map {
      each.location.z = 100 * (each.document.date - (date "1 Jan 1980"))))};
};
```

Another way to handle arrangers is to define them separately in patterns, which are then prefixed to the view later. Because Siri allows multiple prefixing (multiple inheritance) it is easy to use several different arrangers at the same time.

```
TitleByX: aMethod {
   minX, maxX: aNumber;     -- Minimum and maximum X values
```

```
   minX = 0; maxX = 640;

   -- String subtraction is a magnitude computation (0..1).
   pointsOfLight map {
      each.location.x = minX +(maxX-minX)*(each.document.title - "") };
};

DateByY: aMethod {
   firstDate: aDate;
   firstDate.(day = 1; month = 1; year = 1980;)

   pointsOfLight map {
      each.location.y = 100 * each.document.date - firstDate))};
};

AuthorByZ: aMethod {
   minZ, maxZ:     aNumber;        -- Minimum and maximum Z values
   ZGrid:          aNumber;        -- Spacing grid in Z
   allAuthors:     aList;          -- List of author names

   -- Set the minimum and maximum z values.

   minZ = 0; maxZ = 2000;

   -- Compute the allAuthors group.  sorted sorts the group.
   -- asSet takes a group and removes duplicates.

   allAuthors = pointsOfLight map {
      each.document.author, ... } sorted asSet;

   -- allAuthors.length is the size of the list allAuthors.
   -- ZGrid is the gridding in Y between different authors.

   ZGrid = (maxZ-minZ)/allAuthors.length;

   -- Finally, compute the new z locations for all pols.  Use the magic
   -- routine indexOf_in_ to find the author index.

   pointsOfLight map {
      each.location.z = (minZ+
            (indexOf each.document.author in allAuthors)*ZGrid); ...};
};
```

```
-- A complete view, using predefined arrangers.

JimsDocuments2:   Hyperfax, HyperfaxController,
                  TitleByX, DateByY, AuthorByZ {
   observer:   anObserver;
   documents:  aList;

   observer = aStandardObserver;
   documents = HyperfaxDocuments collect { each.author = "Jim Morris" };
};
```

Appendix B: The Junta File

```
--! Loading Junta.Siri...

-- ------------------------------------------------------------------------
--    Junta.siri
--    Siri Primitive Objects
--    Some basic patterns adapted from Bertrand Release
--    by Wm Leler [Leler89].
--    Bruce Horn, last modified 27 October 1993
-- ------------------------------------------------------------------------


--! Fundamental objects----------------------

--! Binding objects to values; bindable slots and bindable objects.
--  Binding integer, numeric, and primitive slots.

"A?anInteger is B'aConstant": aPrimitive { self };
"A'aNumber is B'aNumber":     aPrimitive { self };
"A?any isp B'any":            aPrimitive { self };


--! Contextual evaluation
--  Evaluation of B in the context of A; expressions and variables.

"A'any . [B'any]":            aPrimitive { self };
"A'any . B'aVariable":        aPrimitive { self };


--! Assertion rules-------------------------

"true ; A'any":              { A };          -- TRUE elimination
"(A'any ; B'any) ; C'any":   { A ; B ; C };  -- Parentheses elimination


--! Boolean Operators

"_->_":  anOperator  {  %%precedence is 30; %%associativity is 9;
                        %%opType isp aBoolean; self };
```

```
"_|_":   anOperator  {  %%precedence is 32; %%associativity is 8;
                        %%opType isp aBoolean; self };

"_~|_":  anOperator  {  %%precedence is 32; %%associativity is 8;
                        %%opType isp aBoolean; self };

"_&_":   anOperator  {  %%precedence is 34; %%associativity is 8;
                        %%opType isp aBoolean; self };

"_~&_":  anOperator  {  %%precedence is 34; %%associativity is 8;
                        %%opType isp aBoolean; self };

"~_": anOperator     {  %%precedence is 60; %%opType isp aBoolean;
                        self };


--! Boolean Patterns
-- Distribution of assertion

"(A'any & B'any) ; C'any":    { A ; B ; C };


-- Rewriting to normal forms

"A'aBoolean -> B'aBoolean":   aBoolean { ~(A & ~B) };
"A'aBoolean ~& B'aBoolean":   aBoolean { ~(A & B) };
"A'aBoolean | B'aBoolean":    aBoolean { ~(~A & ~B) };
"A'aBoolean ~| B'aBoolean":   aBoolean { ~A & ~B };
"false & A'aBoolean":         aBoolean { false };
"A'aBoolean & false":         aBoolean { false };
"true & A'aBoolean":          aBoolean { A };
"A'aBoolean & true":          aBoolean { A };
"~true":                      aBoolean { false };
"~false":                     aBoolean { true };
"~~A'aBoolean":               aBoolean { A };


--! Relational Operators

"_<_":   anOperator  {  %%precedence is 50; %%associativity is 10;
                        %%opType isp aBoolean; self };

"_=_":   anOperator  {  %%precedence is 50; %%associativity is 10;
                        %%opType isp aBoolean; self };
```

```
"_>_":   anOperator  {  %%precedence is 50; %%associativity is 10;
                        %%opType isp aBoolean; self };

"_<=_": anOperator   {  %%precedence is 50; %%associativity is 10;
                        %%opType isp aBoolean; self };

"_>=_":  anOperator  {  %%precedence is 50; %%associativity is 10;
                        %%opType isp aBoolean; self };

"_~=_": anOperator   {  %%precedence is 50; %%associativity is 10;
                        %%opType isp aBoolean; self };


--! Relational Patterns------------------------------
-- Rewriting to normal < and <= forms.

"A'aNumber > B'aNumber":   aBoolean { B < A };
"A'aNumber >= B'aNumber":  aBoolean { B <= A };
"A'aNumber ~= B'aNumber":  aBoolean { ~(A = B) };


--! Numeric Objects

aNumericOp:    aNumber;          -- a numeric operator
aNonzero:      aConstant;        -- nonzero constant

aLinear:       aNumber;          -- Linear operator or expression
aLinearExpr:   aLinear;          -- Linear expression
aLinearOp:     aLinear;          -- Linear operator
aLinearTerm:   aLinearExpr;      -- linear term
aSimpleTerm:   aLinear;          -- Simple term


--! Numeric Operators

"-_": anOperator     {  %%precedence is 500; %%opType isp aNumericOp;
                        self };

"_+_":   anOperator  {  %%precedence is 100; %%associativity is 8;
                        %%opType isp aNumericOp; self };

"_-_":   anOperator  {  %%precedence is 100; %%associativity is 8;
                        %%opType isp aNumericOp; self };
```

```
"_*_": anOperator    {  %%precedence is 150; %%associativity is 8;
                        %%opType isp aNumericOp; self };

"_/_": anOperator    {  %%precedence is 150; %%associativity is 8;
                        %%opType isp aNumericOp; self };

"_^_": anOperator    {  %%precedence is 200; %%associativity is 9;
                        %%opType isp aNumericOp; self };

"_++_": anOperator   {  %%precedence is 300; %%associativity is 8;
                        %%opType isp aLinearOp; self };

"_**_": anOperator   {  %%precedence is 350; %%associativity is 8;
                        %%opType isp aLinearTerm; self };

"_^^_": anOperator   {  %%precedence is 400; %%associativity is 8;
                        %%opType isp aLinearOp; self };


--! Numeric Simplification patterns

"0 + B'aNumber":           aNumber { B };
"- B'aConstant":           aNumber { 0 - B };
"- B'aFloat":              aNumber { 0 - B };
"0 * B'aNumber":           aNumber { 0 };
"1 * B'aNumber":           aNumber { B };
"0 / B'aNumber":           aNumber { B~=0 ; 0 };
"A'aNumber ^ 0":           aNumber { 1 };
"A'aNumber ^ 1":           aNumber { A };
"A'aNumber ^^ B'aNumber":  aNumber { A * 10^B };

"A'aNumber + (B'aNumber + C'aNumber)": aNumber { A + B + C };
"A'aNumber * (B'aNumber * C'aNumber)": aNumber { A * B * C };


--! Integer primitives

"A'aConstant < B'aConstant":  aPrimitive  { %%primType isp aBoolean; self };
"A'aConstant <= B'aConstant": aPrimitive  { %%primType isp aBoolean; self };
"A'aConstant = B'aConstant":  aPrimitive  { %%primType isp aBoolean; self };

"A'aConstant + B'aConstant":  aPrimitive  { %%primType isp aConstant; self };
"A'aConstant - B'aConstant":  aPrimitive  { %%primType isp aConstant; self };
```

```
"A'aConstant * B'aConstant":  aPrimitive  { %%primType isp aConstant; self };
"A'aConstant / B'aConstant":  aPrimitive  { %%primType isp aConstant; self };
"A'aConstant ^ B'aConstant":  aPrimitive  { %%primType isp aConstant; self };


--! Floating point primitives

"_asFloat": anOperator { %%precedence is 450; %%opType isp aNumber; self };

"A'aConstant asFloat":  aPrimitive { %%primType isp aFloat; self };

"A'aFloat < B'aFloat":  aPrimitive { %%primType isp aBoolean; self };
"A'aFloat <= B'aFloat": aPrimitive { %%primType isp aBoolean; self };
"A'aFloat = B'aFloat":  aPrimitive { %%primType isp aBoolean; self };

"A'aFloat + B'aFloat":  aPrimitive { %%primType isp aFloat; self };
"A'aFloat - B'aFloat":  aPrimitive { %%primType isp aFloat; self };
"A'aFloat * B'aFloat":  aPrimitive { %%primType isp aFloat; self };
"A'aFloat / B'aFloat":  aPrimitive { %%primType isp aFloat; self };
"A'aFloat ^ B'aFloat":  aPrimitive { %%primType isp aFloat; self };

--! Conversions

-- The following would be easier with true commutativity in the
-- matching process; in that case, we'd only need one of the sets
-- of conversions.


-- Conversions, left constant.

"A'aConstant < B'aFloat":  aBoolean    { A asFloat <  B };
"A'aConstant <= B'aFloat": aBoolean    { A asFloat <= B };
"A'aConstant = B'aFloat":  aBoolean    { A asFloat =  B };

"A'aConstant + B'aFloat":  aFloat      { A asFloat + B };
"A'aConstant - B'aFloat":  aFloat      { A asFloat - B };
"A'aConstant * B'aFloat":  aFloat      { A asFloat * B };
"A'aConstant / B'aFloat":  aFloat      { A asFloat / B };
"A'aConstant ^ B'aFloat":  aFloat      { A asFloat ^ B };


-- Conversions, right constant.

"A'aFloat < B'aConstant":  aBoolean    { A < B  asFloat };
"A'aFloat <= B'aConstant": aBoolean    { A <= B asFloat };
```

```
"A'aFloat = B'aConstant":  aBoolean    { A = B  asFloat };

"A'aFloat + B'aConstant":  aFloat      { A + B asFloat };
"A'aFloat - B'aConstant":  aFloat      { A - B asFloat };
"A'aFloat * B'aConstant":  aFloat      { A * B asFloat };
"A'aFloat / B'aConstant":  aFloat      { A / B asFloat };
"A'aFloat ^ B'aConstant":  aFloat      { A ^ B asFloat };


-- Single argument functions: transcendentals, logarithms, etc.

"sqrt_": anOperator    { %%precedence is 250; %%opType isp aNumericOp; self };
"sin_":     anOperator { %%precedence is 250; %%opType isp aNumericOp; self };
"cos_":     anOperator { %%precedence is 250; %%opType isp aNumericOp; self };
"tan_":     anOperator { %%precedence is 250; %%opType isp aNumericOp; self };
"atan_":    anOperator { %%precedence is 250; %%opType isp aNumericOp; self };
"round_":   anOperator { %%precedence is 250; %%opType isp aNumericOp; self };
"floor_":   anOperator { %%precedence is 250; %%opType isp aNumericOp; self };
"log_":     anOperator { %%precedence is 250; %%opType isp aNumericOp; self };
"log10_":   anOperator { %%precedence is 250; %%opType isp aNumericOp; self };

"sqrt A'aFloat":  aPrimitive { %%primType isp aFloat;    self };
"sin A'aFloat":   aPrimitive { %%primType isp aFloat;    self };
"cos A'aFloat":   aPrimitive { %%primType isp aFloat;    self };
"tan A'aFloat":   aPrimitive { %%primType isp aFloat;    self };
"atan A'aFloat":  aPrimitive { %%primType isp aFloat;    self };
"round A'aFloat": aPrimitive { %%primType isp aConstant; self };
"floor A'aFloat": aPrimitive { %%primType isp aConstant; self };
"log A'aFloat":   aPrimitive { %%primType isp aFloat;    self };
"log10 A'aFloat": aPrimitive { %%primType isp aFloat;    self };


--!Linear Expression Patterns
--! Binding variables

"N?aNumber = K'aConstant ; REST'any":  { N is K ; REST };


--! Creating linear expressions
--  For now, special case constants, floats, and other
--  number objects.

"K'aConstant * V?aNumber": aLinearExpr { K * ((1**V)++0) };
"K'aFloat * V?aNumber":    aLinearExpr { K * ((1**V)++0) };
```

```
"V?aNumber * K'aConstant": aLinearExpr { K * ((1**V)++0) };
"V?aNumber * K'aFloat":    aLinearExpr { K * ((1**V)++0) };

"V?aNumber + N'aNumber":   aLinearExpr { ((1**V)++0) + N };
"N'aNumber + V?aNumber":   aLinearExpr { N + ((1**V)++0) };

"V?aNumber = N'aNumber":   aLinearExpr { ((1**V)++0) = N };
"N'aNumber = V?aNumber":   aLinearExpr { N = ((1**V)++0) };

"V1?aNumber + V2?aNumber": aLinearExpr { ((1**V1)++0) + ((1**V2)++0) };
"V1?aNumber = V2?aNumber": aLinearExpr { ((1**V1)++0) = ((1**V2)++0) };


--! Standard form

"A'aNumber - B'aNumber":      aNumber  { A + (-1 * B) };
"- A'aNumber":                aNumber  { -1 * A };
"A'aNumber / K'aConstant":    aNumber  { (1/K) * A };
"A'aNumber / K'aFloat":       aNumber  { (1.0/K) * A };

"A'nonzero = B'aNumber ; C'any":       { 0 = A - B ; C };
"A'aLinearOp = B'aNumber ; C'any":     { 0 = A - B ; C };
"A?aNumber = B'aNumber ; C'any":       { 0 = A - B ; C };

"LX'aLinear = K'aConstant ; REST'any": { 0 = LX - K ; REST };
"LX'aLinear = K'aFloat ; REST'any":    { 0 = LX - K ; REST };


--! Moving terms together

"A'aLinear + (B'aLinear + C'aNumericOp)":    { (A + B) + C };
"A'aNumericOp + B'aLinear":                  { B + A };
"A'aConstant + B'aLinear":                   { B + A };
"A'aFloat + B'aLinear":                      { B + A };
"A'aLinear * (B'aLinear * C'aNumericOp)":    { (A * B) * C };
"A'aLinear = B'aLinear + C'aNumericOp":      { C = A - B };
"A'aLinear = B'aConstant * C'aNumericOp":    { C = A / B };
"A'aLinear = B'aFloat * C'aNumericOp":       { C = A / B };


--! Nonlinear transformations

"A'aConstant = B'aNumber ^ C'aConstant":     { A^(1/C) = B };
"A'aConstant = B'aNumber ^ C'aFloat":        { A^(1/C) = B };
```

```
"A'aFloat = B'aNumber ^ C'aConstant":        { A^(1/C) = B };
"A'aFloat = B'aNumber ^ C'aFloat":           { A^(1/C) = B };

"A'aConstant = B'aNumber / C'aNumber":       { C~=0 ; B/A = C };
"A'aFloat = B'aNumber / C'aNumber":          { C~=0 ; B/A = C };


--! multiply a linear expression by a constant

"K'aConstant * ((C'aNumber**V'aNumber) ++ REST'aNumber)":
      { ((K*C)**V) ++ (K * REST) };

"K'aFloat * ((C'aNumber**V'aNumber) ++ REST'aNumber)":
      { ((K*C)**V) ++ (K * REST) };

"((C'aNumber**V'aNumber) ++ REST'aNumber) * K'aConstant":
      { ((K*C)**V) ++ (K * REST) };

"((C'aNumber**V'aNumber) ++ REST'aNumber) * K'aFloat":
      { ((K*C)**V) ++ (K * REST) };


--! add a constant to a linear expression

"K'aConstant + ((C'aNumber**V'aNumber) ++ REST'aNumber)":
      { (C**V) ++ (K + REST) };

"K'aFloat + ((C'aNumber**V'aNumber) ++ REST'aNumber)":
      { (C**V) ++ (K + REST) };

"((C'aNumber**V'aNumber) ++ REST'aNumber) + K'aConstant":
      { (C**V) ++ (K + REST) };

"((C'aNumber**V'aNumber) ++ REST'aNumber) + K'aFloat":
      { (C**V) ++ (K + REST) };


--! add two linear expressions

"lxMerge_": anOperator  {  %%precedence is 30; %%opType isp aLinearExpr;
                           self };

"_lexc_":   anOperator  {  %%precedence is 50; %%associativity is 10;
                           %%opType isp aNumber; self };
```

```
"V1?aNumber lexc V2?aNumber": aPrimitive { %%primType isp aNumber; self };

"((C1'aNumber**V1?aNumber)++R1'aNumber) +
 ((C2'aNumber**V2?aNumber )++R2'aNumber )":
      { lxMerge ( (V1 lexc V2), ((C1**V1)++R1), ((C2**V2)++R2)) };


-- First variable sorts first
"lxMerge( 0, (T1'aNumber++R1'aNumber) , LX2'aLinear )":
      { T1 ++ (R1 + LX2) };


-- Second variable sorts first
"lxMerge( 2,      LX1'aLinear,
                  (T2'aNumber++R2'aNumber))":
      { T2 ++ (LX1 + R2) };


-- Variables equal
"lxMerge( 1,      ((C1'aNumber**V1?aNumber)++R1'aNumber) ,
                  ((C2'aNumber**V2?aNumber)++R2'aNumber))":
      { lxMerge ( (C1 = -C2), ((C1**V1)++R1) , ((C2**V2)++R2)) };


-- Merging opposite coefficients
"lxMerge( true,   (T1'aNumber++R1'aNumber),
                  (T2'aNumber++R2'aNumber) )":
      { R1 + R2 };


-- Merging inequal coefficients
"lxMerge( false,  ((C1'aNumber**V1'aNumber)++R1'aNumber) ,
                  ((C2'aNumber**V2'aNumber)++R2'aNumber) )":
      { ((C1+C2)**V1) ++ (R1 + R2) };


--! solving
-- We simply take the first variable in the ordered expression and solve
-- for it.  Interesting variables a la Bertrand are not used.

"0 = ((C'aNumber**V?aNumber) ++ REST'any) ; EX'any":
      { (V is ((-1/C) * REST)) ; EX };
```

```
--! cleaning up after a bound variable
-- A bound variable can be replaced by a constant or a linear.
-- Handle constants and floats explicitly.

"(C'aConstant ** K'aConstant) ++ REST'any":  { (C*K)  + REST };
"(C'aConstant ** K'aFloat) ++ REST'any":     { (C*K)  + REST };
"(C'aConstant ** LX'aLinearOp) ++ REST'any": { (C*LX) + REST };

"(C'aFloat ** K'aConstant) ++ REST'any":     { (C*K)  + REST };
"(C'aFloat ** K'aFloat) ++ REST'any":        { (C*K)  + REST };
"(C'aFloat ** LX'aLinearOp) ++ REST'any":    { (C*LX) + REST };


--! Debugging and I/O

"prefixOf_": anOperator { %%precedence is 25; %%opType isp anObject; self };
"_print":   anOperator  { %%precedence is 25; %%opType isp aBoolean; self };

"prefixOf A'any":    aPrimitive { self };
"A'aString print":   aPrimitive { self };
"A'any print":       aPrimitive { self };


--! Finished loading Junta.siri
-- Junta.siri END-------------------------------------------------------------
```